08/Jan/2019

- Searching can be done on probability basis.
- Binary Search = Sort + search.
- Binary search is based on "Divide and conquer" strategy.
- Linear search - Iterative.
- Greedy approach - It first search which has the highest probability.

Step 1 : problem statement (well known or scenario based)

Step 2 : Category of problem (P, NP)

Step 3 : strategy of algorithm design.

1. Iterative (P)
2. Divide and Conquer (P)
3. Backtracking (NP)
4. Branch and Bound (NP)
5. Greedy strategy (P)
6. Dynamic programming (P)
7. Recursion (P)
8. Recursion with backtracking (NP)

NOTE: For NP problems Divide and conquer cannot be used.

Step 4 : Algo writing (plain english) and chick the correctness of algo

worst	Best	sort	strategy
n^2	$n\log_2 n$	Quick sort	Divide and Conquer
2^n	$n\log_2 n$	Merge sort	" " "
n^2	n	selection sort	Iterative
n^2	n	Insertion sort	"
n^2	n/n^2	Bubble sort	"

n = size of data,

eg. int arr[10];
$n = 10$

Step 5 : Asymptotic Complexity analysis of algo.
① time
② space
③ Network

Ques. How to lowere down n/w complexity?
The validation is done at the client side and the final data will be send to the server then the server will send a small message.

- Asymptotic analysis :- means how much time space will be required by the algo based on the size of the data.

NOTE: more nested loops leads to more complexity

Correctness - which terminates and which gives correct o/p on I/P.

Unit 1 - Step 4 Step 5	
Unit 2, 3 - Step 3	
Unit 4 - Step 2	

Step 5 : Asymptotic analysis algorithm

Time complexity :

algo1:
```
int arr[20]
for (i=1; i<n; i++)          — (n)
{
   pf (arr[i])
}
for (i=1 to 2)               — (2)
  pf(i)
for (i=n; i>1; i--)          — (n)
  pf (arr[i])
```

algo2:
```
int arr[20][20]              — (1)
for i=1 to n                 — (n)
{   for j=1 to n             — (n)
    {
        pf (arr[i][j])
    }
}
```

algo 1 : $n = 20$

instruction $= 2n+2 = 42$

Complexity $= 2n+2$
$= n$

If data is of size 'n' then $(2n+2 \simeq n)$ instructions will be executed to work on data.

algo 2 : time complexity $= n^2+1$ or n^2

$i=1 \rightarrow j=1$ to n
$i=2 \rightarrow j=1$ to n
$\vdots$
$i=n \rightarrow j=1$ to n

NOTE: (1) In nested loops - multiply

(2) In independent loops - addition

09/Jan/2019

UNIT - I

* Asymptotic Analysis :-

1. Time
2. Space (compromised)
3. Network

Asymptotic time analysis or asymptotic time complexity of any algo. specifies the no. of instructions which will be executed to process 'n' size data.

Time → no. of instructions executed ∝ n (data size)
by algo

Time Complexity of Insertion sort :-

```
int arr[n]                  — ① time
for i = 2 to n              — (n-1) times
{
    Key = arr[i]            — once for each (n-1)
    j = i - 1               — once for each (n-1)
    while (j>0 && Key < arr[j])
    {
        arr[j+1] = arr[j];  — once for each j
        j = j-1;
    }
    arr[j+1] = Key          — once for each i
}
```

```
        j = j-1;
    }
    arr[j+1] = key
}
```

eg. for i = 2 to n-1 $\longrightarrow n-2$

```
{
    for j = 2 to n-1   ⟶ n-2
    {
        pf (i*j)
    }
}
```

Independent loops

Time Complexity $= (n-2) * (n-2)$

$= n^2 - 4n + 4$

$\simeq n^2$

Time complexity of Insertion sort :

1st method $\underbrace{(n-1)}_{i} * \underbrace{(n-1)}_{j\ (max)} = n^2 - 2n + 1$

$\simeq n^2$

for i=2, j=1 ; for i=3, j=2

2nd method $1 + 2 + 3 + 4 + 5 + \underbrace{6}_{n-1}$

= Sum of (n-1) natural no.

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

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

$$= \frac{n^2 - 2n + 2}{2} \simeq n^2$$

Difference b/w 2 methods :-

(1) In first we assumed that j will execute 6 times but in second method we are exactly adding how many times j will be executed for every i.

(2) eq'n difference

eg. $n = 7$

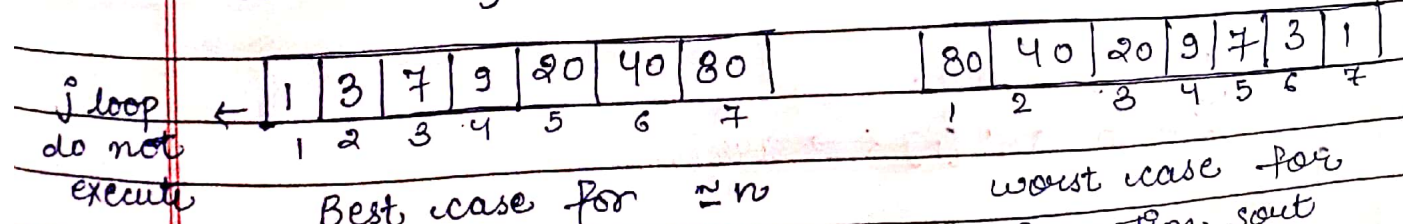
1st method - $49 - 14 + 1 = 36$

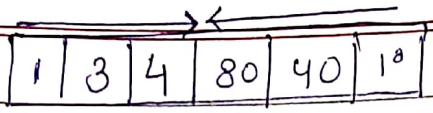
2nd method -

$$\frac{49 - 14 + 2}{2} = 18.5$$

* Effects of dataset on the algorithm time Complexity?

Algorithm time complexity depends upon the configuration of data (i.e data is sorted, unsorted, in ascending order, descending order)





avg. case for insertion sort.

eg. | 3 | 9 | 17 | 80 | 40 | 12 | 2 |

① 3 – 1st time (Best case of algo)

② 2 – 7th time (Worst case algo)
= nth time

③ 80 – 4th time (avg case for algo)
= $\frac{n}{2}$ time

15/Jan/2019

* **Asymptotic Analysis of Selection Sort :-**

```
Selection sort ()
int arr[n];          → insert element in arr[i]
for i=1 to n-1
{
    min = i
    for (j= i+1 to n)
    {
        if ( arr[min] > arr[j])
        {
            min = j;
        }
    }
    if i != min
    Swap arr[i] ↔ arr[min]
}
```

→ Best case : (data sorted)

8	9	12	15	16	19	80

i (min), j

i = 1 j = 2 to 7
i = 2 j = 3 to 7
i = 3 j = 4 to 7
i = 4 j = 5 to 7
i = 5 j = 6 to 7
i = 6 j = 7 to 7

NOTE : min. is not reset.

→ Worst case. : (data is sorted in reverse order)

80	19	16	19	12	9	8

i, j

min = i

NOTE : min is getting updated.

→ Time complexity (best case) $\simeq n^2$ or $\simeq n$

→ Time complexity (worst case) $\simeq n^2$

→ Average case :

For selection sort which is implemented to sort elements in ascending order we have 3 cases of data :-

Case 1 : data sorted in ascending order (best case)

Case 2: data sorted in descending order (worst)

In both the cases 'i' loop will execute (n-1) times and 'j' loop will execute from i+1 to n as follows.

i=1 j=2 to 7
i=2 j=3 to 7
⋮ ⋮

In both worst and best case data time complexity of the algorithm is

$= (n-1) * (n-1)$
for i ↙ ↘ for j (max iterations of j for i)

$= n^2 - 2n + 1$
$\simeq n^2$

But in best case data, the min. variable will never be reset and for the worst case min. variable will be reset everytime 'j' loop is executed.
So because of the overhead it could be said that the best case complexity is 'n' from the min. point of view.

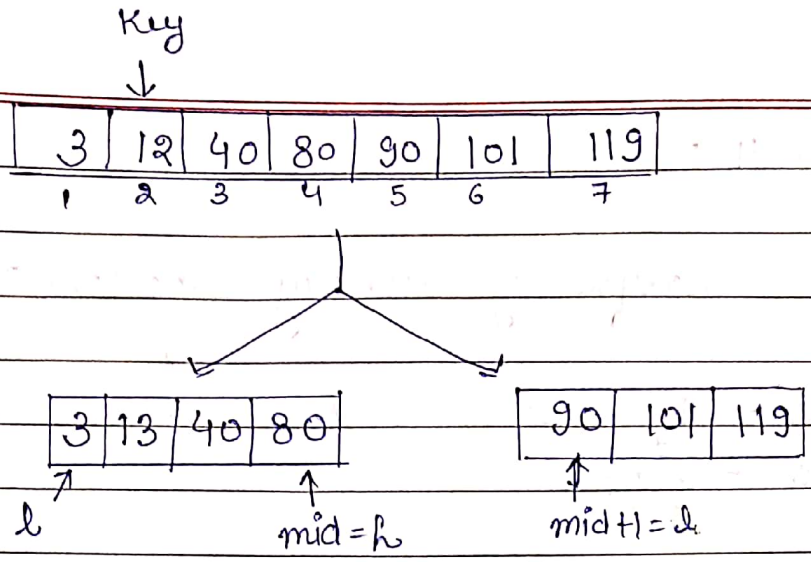
* BINARY SEARCH :

→ Asymptotic analysis of binary search :-

① Best case - 1
② Worst case - $\log_2 n$

3	90	40	13	80	101	119
1	2	3	4	5	6	7

```
Binary search (l, h, Key)
{
    int mid;
    while (l <= h)
    {
        mid = (l+h)/2;
        if ( Key == arr[mid])
        {
            print exit();
        }
        else if ( Key < arr. [mid]
        {
            h = mid;
        }
        else
        {
            l = mid + 1;
        }
    }
}
```



$n \simeq 2^i$ → array is divided in 2 parts

$7 \simeq 2^3$

↳ taking $\log_2$ $\log_2 n = \log_2 2^i$

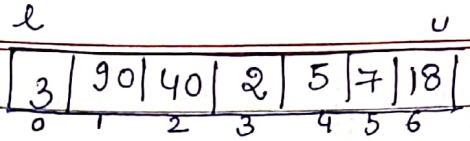
$\Rightarrow \log_2 n = i * \log_2 2$

$\boxed{P = \log_2 n}$ n = array size

NOTE: If the strategy is divide and conquer then complexity will be $\log n$.

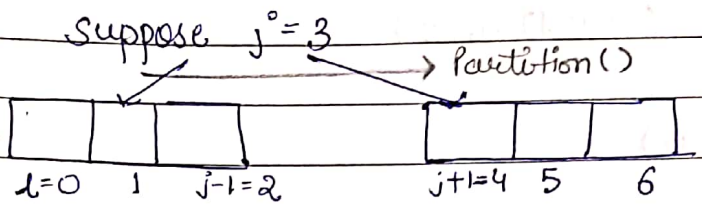
* Asymptotic analysis of Quick sort :

```
Quick sort (arr[], l, u)
{
    if (l < u)
    {
        j = partition (arr, l, u)        // j → index at which we are dividing the array
        Quick sort (arr, l, j-1)
        Quick sort (arr, j+1, u)
    }
}

Partition (arr[], l, u)
{
    Key = arr[l]
    i = l+1
    j = u
    while (i<j)        (n-1) times ≃ n
    {
        while (arr[i] < key)    i++;

        while (arr[j] > key)    j--;

        if (i<j)
            swap arr[i] ↔ arr[j]
    }
    if (i>j)
        swap arr[l] ↔ arr[j]

    return j
}
```

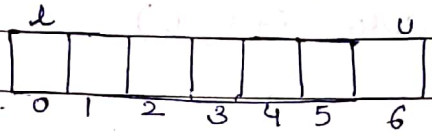


eg.

When we do partition everytime the value of i value is lost

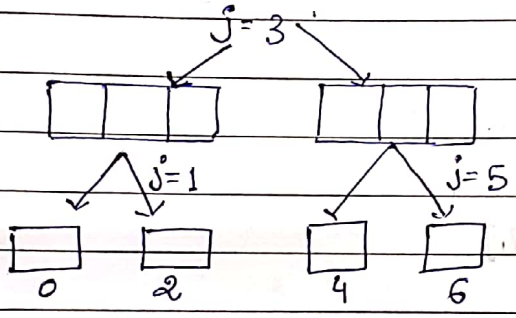


① Best case data for Quick sort :

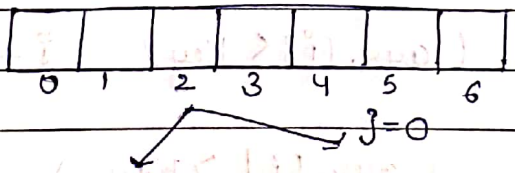


i QS

$i = \log_2 n$



② Worst case :

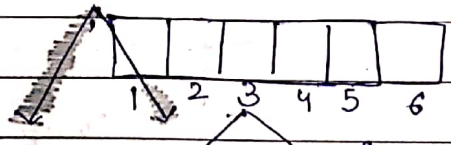


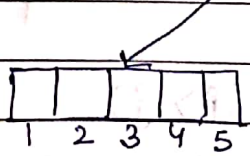
l to j-1

0 -1

this will not exist always

j+1 to u





j+1 = 7

n = 7 QS

i = n = 7

5 times quick sort will be called

NOTE: $n = 2^i$

$i = 2^3$

In best case the array will be sorted in 3 times but in worst case the no. of times quick sort called is 'n' times (i.e data)

best case, $i = \log_2 n$

worst case, $i = n$

→ Best case: array with elements where partition is in mid.

Best case = $\underbrace{\log_2 n}_{\text{QS recursion}} * \underbrace{n}_{\text{partition}}$

$\simeq n \log_2 n$

→ Worst case:

Worst case = $\underbrace{n}_{\text{QS recursion}} * \underbrace{n}_{\text{partition}}$

$\simeq n^2$

In worst case array is sorted and the partition takes place at the first or the last position.

Worst case

2	3	9	13	40	60	80
0	1	2	3	4	5	6
(l)						(u)

Key = arr[l]
= 2

i = l + 1 = 1
j = u = 6

while (i < j)
while (arr[i] < Key)
arr[1] ≮ Key
3 2

while (arr[j] > Key)
arr[j] > Key
80 > 2
60 > 2 j = 5
40 > 2 j = 4
if (i < j) 13 > 2 j = 3
swap arr[i] ↔ arr[j]

if (i ≥ j)
swap (arr[l] ↔ arr[j])

return j;
j = 0

Best case

13	9	3	2	80	40	60
0	1	2	3	4	5	6
(l)						(u)

Key = arr[l]
= 13

i = l + 1 = 1
j = 6

while (i < j) ✓
while (arr[i] < Key)
i = 1, arr[1] < Key
9 < 13

i = 2, arr[2] < Key
3 < 13

i = 3, arr[3] < Key
2 < 13

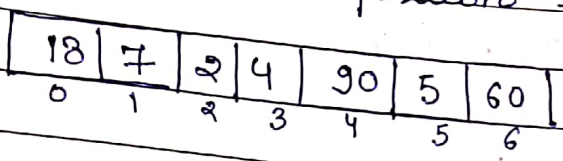
i = 4, arr[4] < Key
80 < 13 ✗

4 3
if (i < j)
swap arr[i] ↔ arr[j]

if (i ≥ j)
swap

j = 3 (return)
0-2 4-6

Ques: In how many situations, element 18 would be added in actual position.



NOTE: In merge sort sorting takes place at Merge step.

* MERGE SORT:

LB, mid = (LB+UB)/2, UB = 6

```
Merge (arr[], p, q, r)
{
    n1 = q - p + 1
    n2 = r - q

    L[1 to n1+1]
    R[1 to n2+1]

n/2 ≃ n1 times ← for i = 1 to n1
        L[i] = arr[p + i - 1]

n/2 ≃ n2 times ← for j = 1 to n2
        R[j] = arr[q + j]

    i = 1
    j = 1
    for K = p to r
    {
        if (L[i] ≤ R[j])
        {
            arr[K] = R[j]
            j = j + 1
        }
        K++
    }
}
```

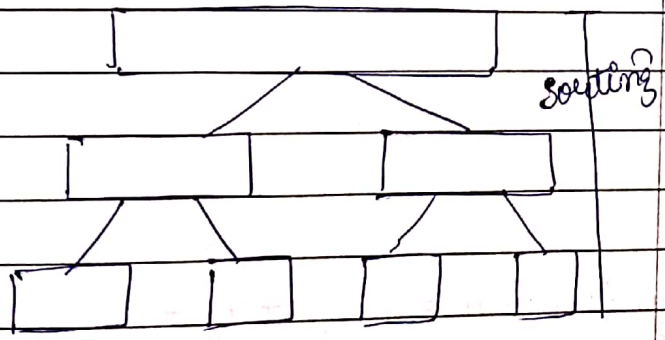
n_1 = size of left subproblem

n_2 = " " right "

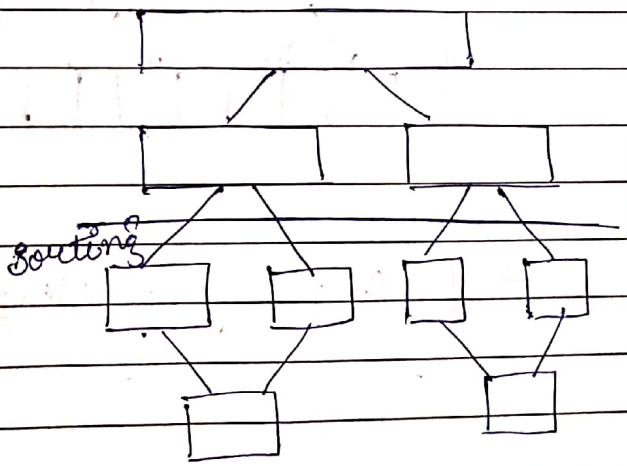
```
MS( arr, p, r)
{
    if (p<r)
    {
        mid = (p+r)/2
        MS (arr, p, mid)
        MS (arr, mid+1, q)
        Merge (arr, p, q, r)
                    "mid.
    }
}
```

→

Quick sort



Merge sort



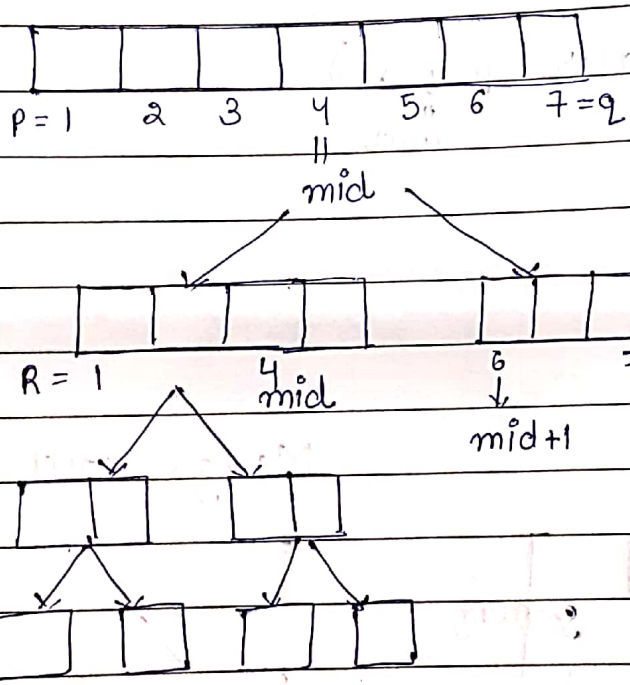
$n = \log_2 n$ = Best = Worst

~~best~~ worst or ~~worst~~ data can be any arrangement of data

Asymptotic time complexity of Merge Sort = No. of times M.S call ($\log_2 n$) X time taken by Merge each time (n)

$= n \log_2 n$

eg.



$mid = \frac{1+7}{2} = 4$

$n = 2^i$

$7 = 2^3$

→ Complexity of Merge function = $\frac{n}{2} + \frac{n}{2} + n$

$= \frac{n+n+2n}{2} = \frac{4n}{2} = 2n$

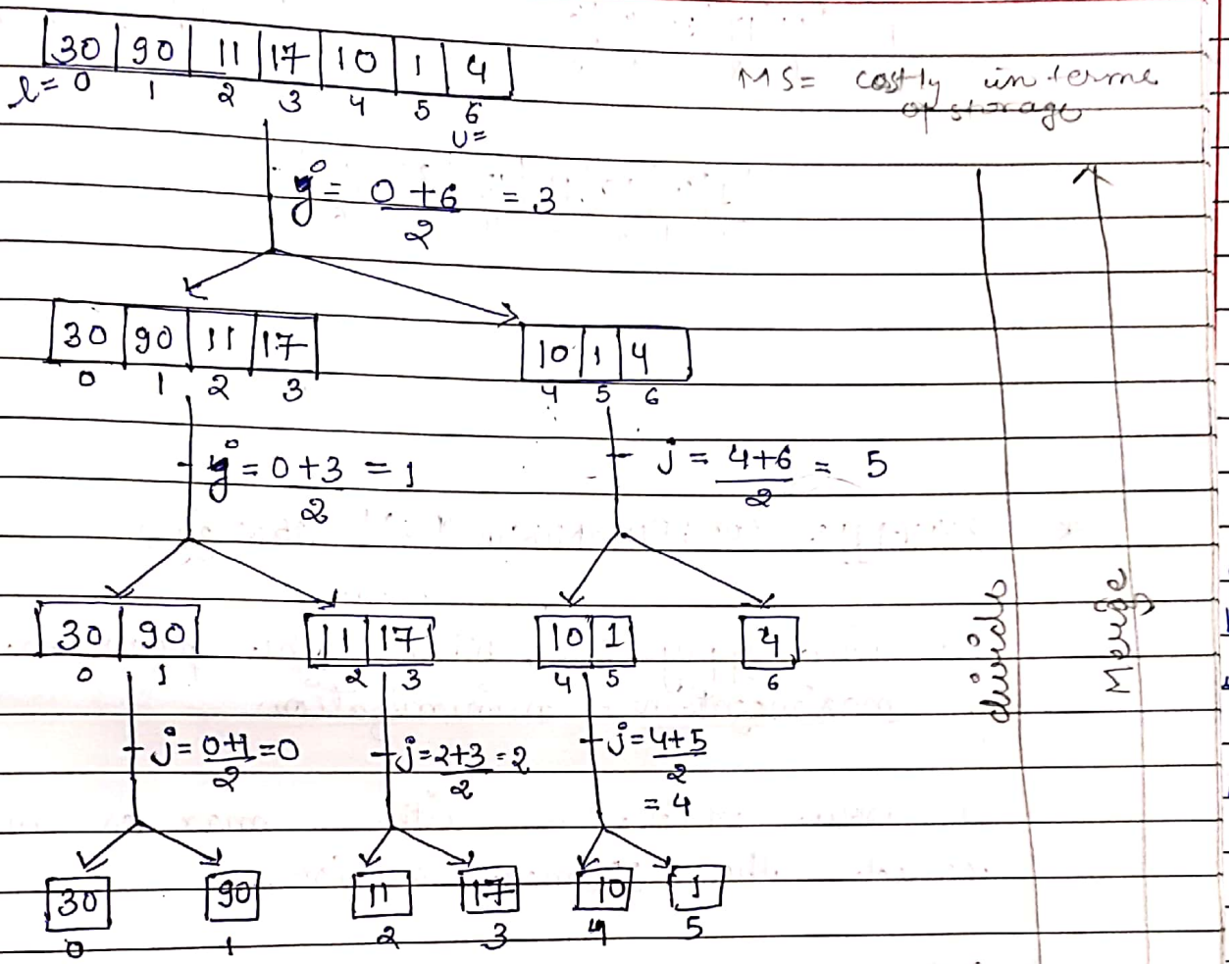
$\simeq n$

HW Ques.

NOTE: Selection and Insertion - Iterative strategy
Binary, Merge, Quick - Divide and conquer strategy
* not in place sorting = MS

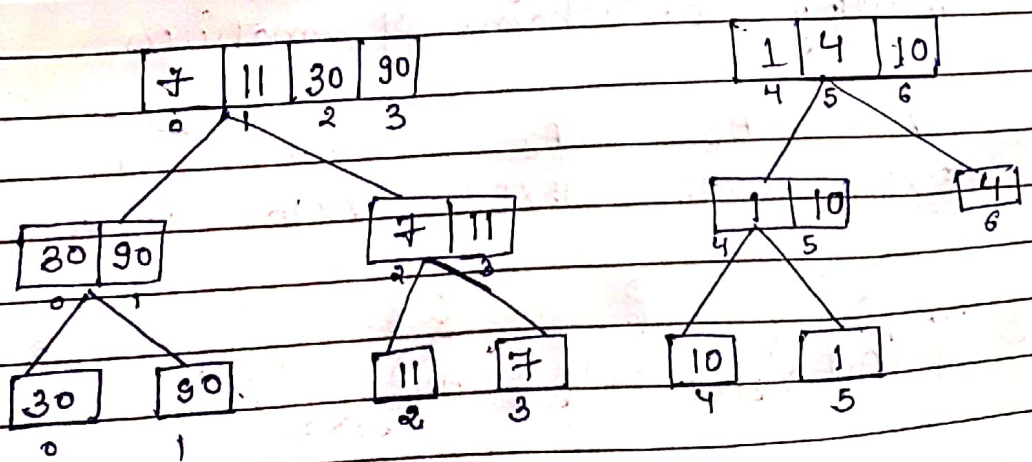
25/Jan/2019

Ques.



MS = costly in terms of storage

Now the merge function is called.



NOTE: for matrix multiplication, the no. of columns of first matrix and no. of rows must be same

L[1 to 5]

	1	2	3	4	5
L	11	17	30	90	∞

R = [1 to 4]

	1	2	3	4
R	1	4	10	∞

0	1	2	3	4	5	6
1	4	10	11	17	30	90

$n_1 = q - p + 1 = 3 - 0 + 1 = 4$

$n_2 = r - q = 6 - 3 = 3$

* DYNAMIC PROGRAMMING (3rd Strategy)

or greedy

We will apply DP when the problem is of maximization or minimization.

A solu. which is either max or min is called the optimal solution.

DP is called cumbersome becz it checks all methods.

(Problem)

• Matrix Chain Multiplication :-

Ques. Find min. of multiplications required to multiply a chain of matrices

A_1	A_2	A_3	A_4	A_5	$A_6 = n_6$
30X35	35X15	15X5	5X10	10X20	20X25

row col

① $((((A_1 A_2)_{30\times15} A_3)_{30\times5} A_4)_{30\times10} A_5)_{30\times20} A_6)_{30\times25}$

(30X35, 35X15, 15X5, 5X10, 10X20, 20X25)

② Answer: $(A_1(A_2(A_3(A_4(A_5 A_6)_{10\times25})_{5\times25})_{15\times25})_{35\times25})_{30\times25}$

③ $(((A_1 * A_2)(A_3 * A_4))(A_5 A_6))$ or more

eg.

A_1	A_2	A_3	A_4
2×3	3×5	5×1	1×7

① $((A_1 A_2)(A_3 A_4))$ case 1
② $(((A_1 A_2) A_3) A_4)$ case 2
} cost = ?

Case 1 :

$2\times3 \quad 3\times5 \quad 5\times1 \quad 1\times7$

$((A_1 A_2)(A_3 A_4))$ 2×7
$2\times5 \quad 5\times7$

```
for i = to 2 ——— i * 3 * 5
{
   for j = 1 to 3
   {
      for k = 1 to 5
      {
      }
   }
}
```

$A_1A_2 = 2*3*5 = 30$
$A_3A_4 = 5*1*7 = 35$
$(A_1A_2)(A_3A_4) = 2*5*7 = 70$

cost = 30 + 35 + 70 = 135 multiplications

NOTE: no. of multiplication to multiply 2 matrices, $A_{i \times j}$ & $A_{j \times k}$
$= i * j * k$

Case 2:

NOTE: In case of DP, we whenever calculate solution to any subproblem it is safe to use further while calculating solution to another problem, this concept is called as Memoization.

$(A_1 A_2)_{2 \times 5} = 30$

$(A_1 A_2)_{2 \times 5} A_3)_{2 \times 1} \Rightarrow 2 \times 5 \times 1 = 10$

optimal sol. structure $\leftarrow \{(A_1 A_2) A_3\}_{2 \times 1} * A_4) = 2 \times 1 * 7 = 14$

(optimal sol) Total cost = $30 + 10 + 14 = 54$ multiplications
↓ optimal sol. value

NOTE: Matrix multiplication problems ask for the order of the matrix multiplication keeping the sequence of matrices same in which cost of multiplication of matrix chain is minimum.

01/Feb/2019

To multiply chain of matrices in given sequence through one order which results into min. no. of multiplication.

- 4 steps to solve DP problem:-

Step 1: Optimal substructure

Step 2: Recursive soln.

(algo) Step 3 : optimal soln value

Step 4 : optimal soln structure (find out optimal paranthesization structure)

eg. $A_1 \dots A_n$ (Step 1)

If we get optimal soln from A_1 to A_n, so we can get optimal soln for all the substructures of A_1 to A_n will be optimal.

$A_1 \quad A_2 \quad A_3 \quad \dots\dots \quad A_6$

optimal soln → (min) 500 $(A_1 A_2)((A_3 A_4)A_5)A_6))$ option 1

650 $(((A_1 A_2)(A_3 A_4))(A_5 A_6))$ option 2

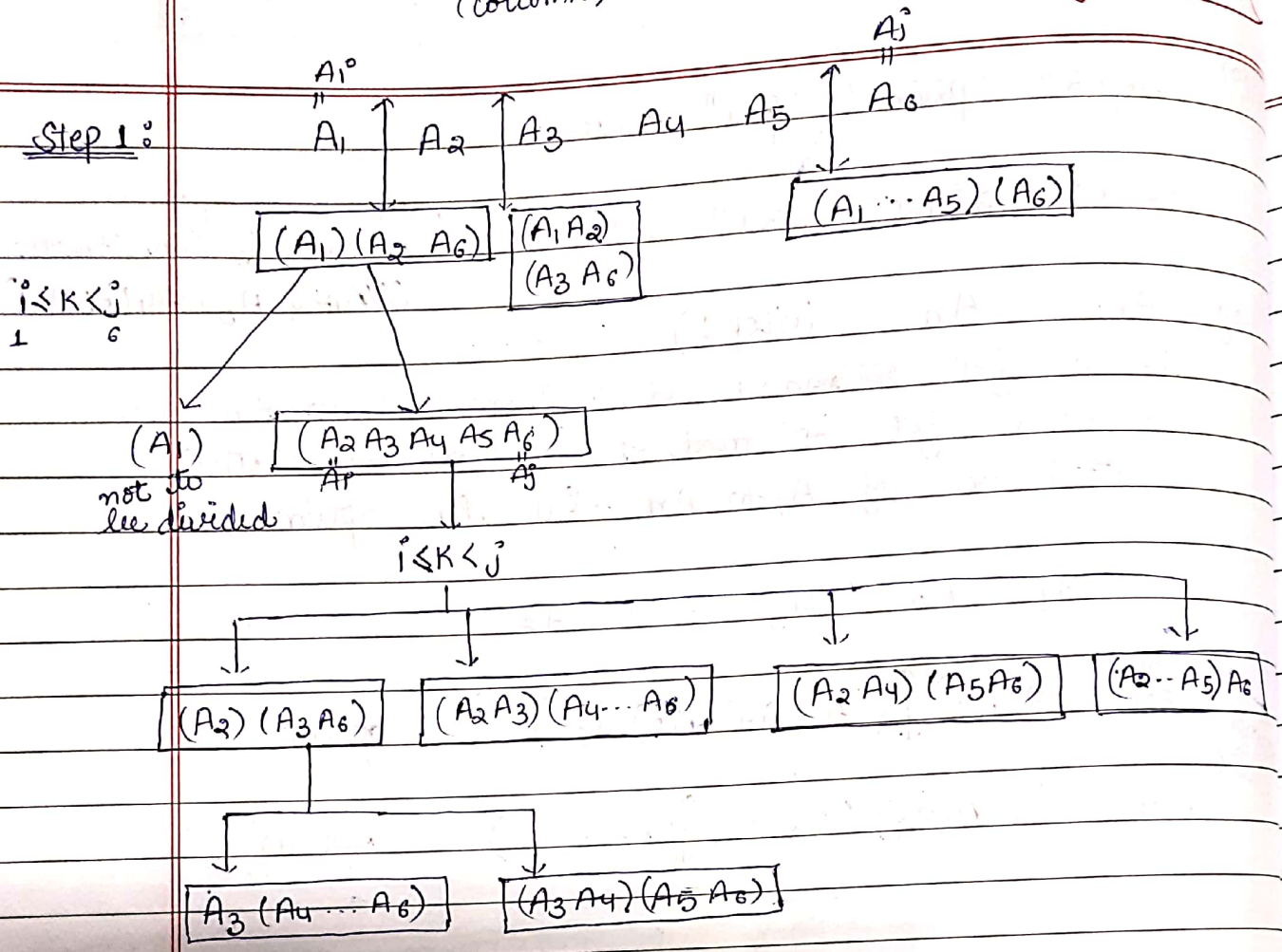
Step 1 : In case of DP, the problem should have optimal substructure property which means that if a particular parenthesization gives optimal soln to the problem that means the soln to its subproblem is also optimal.

Step 2 : In this step we write down the recursive soln to the problem.

(algo) gives a no. Step 3 : The algorithm gives us the optimal soln value for the problem.

(algo) gives a structure Step 4 : It shows the actual parenthesization through which we get optimal soln in Step 3.

p * j * K → (column in 2nd)
(row) ↓
(column)



In step 1, we divide the given problem Ai to Aj at any point 'K', where K ranges from $i \leq K < j$ and we get two subproblems i.e $(A_i \cdots A_K)$ and $(A_{K+1} \cdots A_j)$, which need to be divided further till every subproblem consists of single matrix bcz the cost of multiplication of single matrix is zero.

Step 2: Recursive soln

$$m[i,j] = \begin{cases} 0 & i=j \quad \text{(only 1 matrix in matrix chain)} \\ \min_{i \leq K < j} \{ m[i,K] + m[K+1,j] + i*j*K \} & \text{if } i<j \end{cases}$$

m[i,K]: left subproblem sol. ; m[K+1,j]: Right subproblem sol.

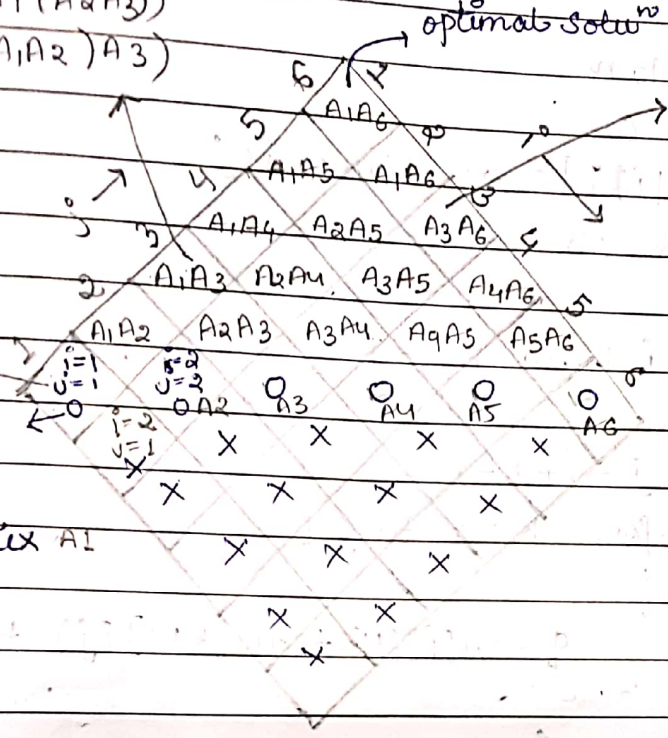
2D matrix to store optimal sol^n value for all problem and subproblem

m[i,j] stores optimal no. of multiplications required to multiply matrices A_i to A_j (ie A_1 to A_6)

Two possibilities from A_1 to A_3 (where $k=1,2$)

① $(A_1(A_2A_3))$

② $((A_1A_2)A_3)$



m[i][j]

Cost of multiplication of single matrix A_i = 0

we have to check where j=i or i

NOTE:
we do not consider i>j case.

06/Feb/2019

Step 3 :

optimal Soln Value ⇒ min no of multiplication required to multiply chain of matrices $A_i \cdots A_j$

$A_1 \cdots A_n$

algo
- m[n][n] (n=6) → From this matrix we'll get the optimal Soln value.
- s[n][n]

for i = 1 to n
m[i][i] = 0 → all m[i][j] = ∞ where i≤j
for d = 2 to n
{
for i = 1 to n-d+1

Optimal solution value () suppose $(n=6) \rightarrow A_1 \cdots A_6$

m[n][n]

s[n][n]

For i = 1 to n

m[i][i] = 0

all m[i][j] = ∞ where i<j

(n-1) ← for l = 2 to n

{

n ← for i = 1 to n-l+1

{

j = i+l-1

n ← for K = i to j-1

{

q = m[i][K] + m[K+1][j] + i*j*K

if q < m[i][j]

{

m[i][j] = q (m[1][n] gives value of optimal soln)

s[i][j] = K

}

}

}

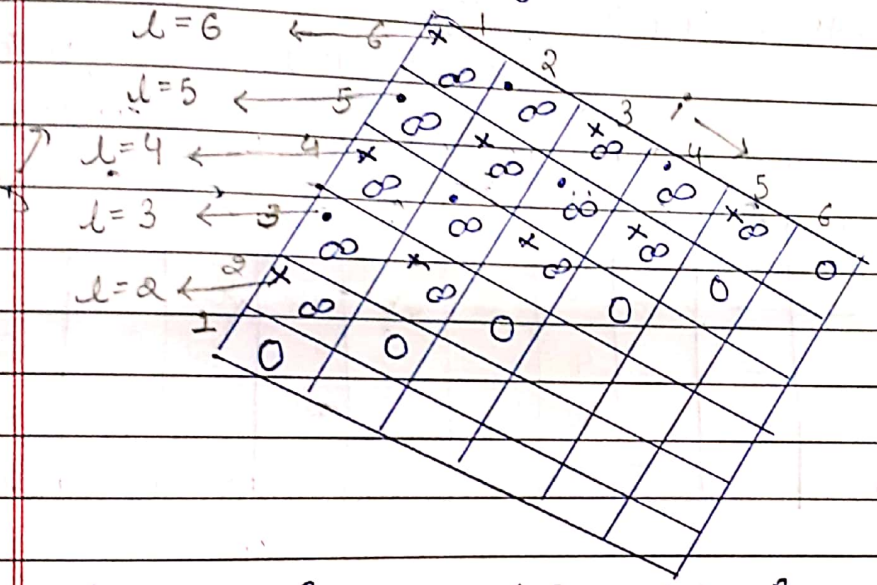
}

NOTE: For every algorithm consider these 4 steps :-

1. algo
2. dry runs
3. time complexity
4. correctness.

NOTE: ① The 2D array 'm' stores the min. no. of multiplications required to multiply any problem or subproblems from Ai to Aj at index m[i][j]

② The array 's' stores the optimal division point K for the problem Ai to Aj and this 'K' resulted into min. no. of multiplications required for the chain of matrices Ai to Aj.



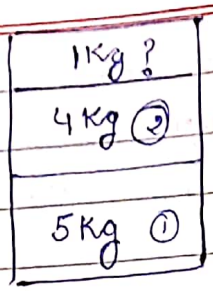
l = 2 i = 5 (for l=2, i executes 5 times)
l = 3 i = 4
l = 4 i = 3
l = 5 i = 2
l = 6 i = 1

07/02/2019

Problem 2: Knapsack problems → maximization → 0/1 (DP) } greedy
→ fraction }

Items → i	1	2	3	4
weight	5	4	6	3
profit (Rs)	10	40	30	50

Bag = 10 kg.



0/1 Knapsack:

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

< 10Kg & max value

fractional Knapsack:
we can pick any part of the item.

2+4 ⇒ 4+3 ⇒ 7Kg.
40+50 = 90 Rs

Knapsack
w=10Kg

	1	2	3	4	5	6	7	8	9	10
i=1 (1)	—	—	—	—	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)	10 (1)
i=2 (1,2)	—	—	—	40 (2)	40	40	40	40 (2)	50	50
i=3 (1,2,3)	—	—	—	40 (2)	40	40	40	40	50	70
i=4 (1,2,3,4)	—	—	50 (4)	50 (4)	50	50	90 (2+4)	90 (2+4)	90	90

value has to be max.

Can we make any combination of 1 and 2 to make a weight of 10Kg

pick (max value) ✓

10 (1) 5
40 (2) 4

10 (1)
40 (2)
50 (1,2)

Optimal solution = 90 (2+4)

14/Feb/2019

minimize cost of search of keys in BST

minimization

Numerical

Problem 3 : optimal Binary Search Tree. (OBST)

English (source lang) → Translator (S/w) → German (Target lang)

eg. This is a nice book.

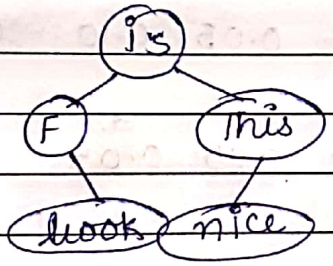
Eng	german
This	XY
is	F
a	AB
nice	
book	

array or linked list

linear search (n)

↓

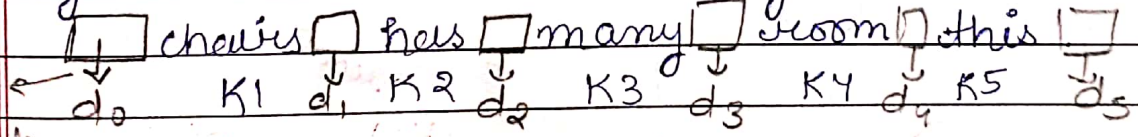
no. of keys supported by translator



binary search ($\log_2 n$)

($\log_2 n < n$)

eg. This room has many chairs.



words which are not here

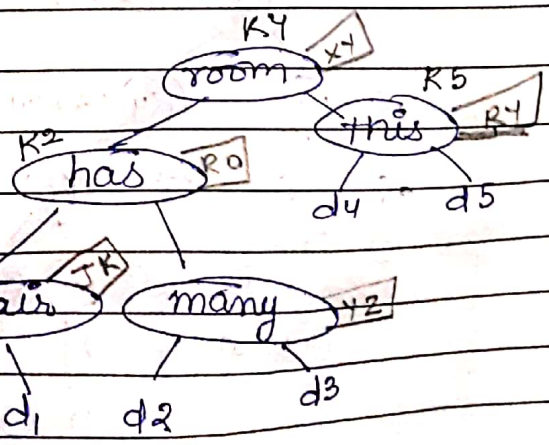
successful

K_i = Key

d_i = dummy key

↓

unsuccessful.



Consider one case in paper & find its cost

OBST problem is to find a BST based on 'n' keys such that the cost of searching any key in BST is minimal provided p_i and q_i where

p_i = probability of occurrence of any key K_i ~~and~~

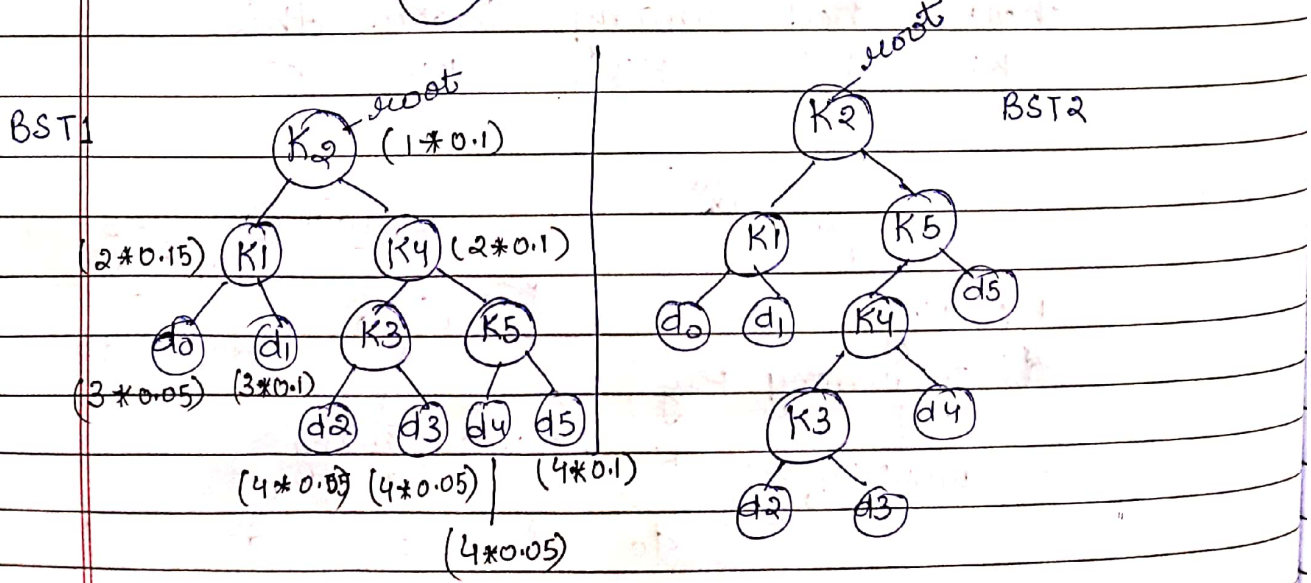
and q_i = probability of occurrence of any dummy key d_i

n = 5 Keys supported by translator

p_i	K_i	— Key
q_i	t_i	— dummy key

i	0	1	2	3	4	5
		K_1 (chair)				
p_i	—	0.15	0.10	0.05	0.10	0.20
	d_0				d_3	
q_i	0.05	0.10	0.05	0.05	0.05	0.10

This is my chair
d_3



Cost of searching in BST $= \sum_{i=0}^{n} (d_i + 1) * q_i + \sum_{i=1}^{n} (K_i + 1) * p_i$

d_i → depth → dummy Key

K_i → depth → Key

$K_1 = 0.3$	
$K_2 = 0.1$	
$K_3 = 0.15$	
$K_4 = \cancel{0.15}\ 0.2$	
$K_5 = \cancel{0.2}\ 0.6$	
$d_0 = \cancel{0.6}\ 0.15$	
$d_1 = 0.15\ 0.3$	
$d_2 = \cancel{0.3}\ 0.2$	
$d_3 = 0.2$	
$d_4 = 0.2$	
$d_5 = 0.4$	
2.80	2.75

5/6/2019

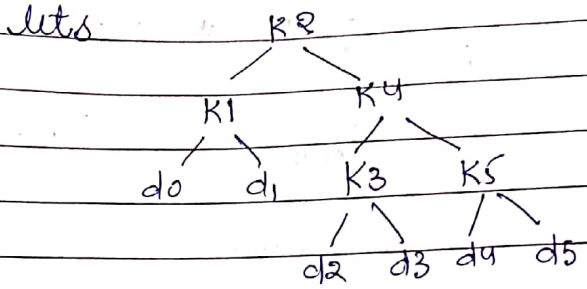
OBST :

Step 1 : optimal substructure

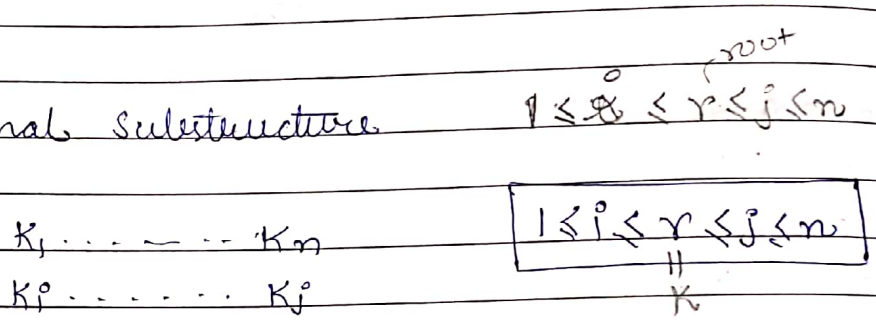
Step 2 : Recursive soln

Step 3 : optimal soln value (Algo) — gives optimal cost of searching in a BST of given n keys

Step 4 : optimal soln structure (Algo)
algo for optimal BST structure.



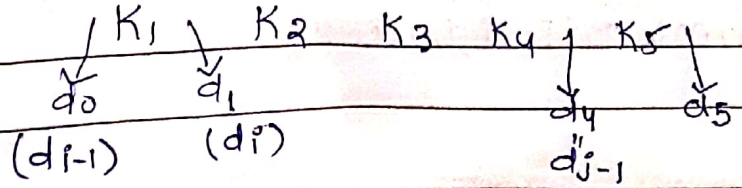
Step 1: optimal substructure



NOTE: DP is Bottom-up approach.

OBST problem ensures optimal substructure property because if we get a solution of the $\hat{}^{min}$ searching cost for the actual problem $K_1 \cdots K_n$ that means we have the optimal searching cost for any of the subproblem

Step 2: Recursive soln



optimal BST from k_3 to k_6

$e[3,6]$ = Suppose (2.6)

isme cost store hogi cost BST ki

$e[i,j]$ = contains the optimal searching cost in BST created on keys $k_i \dots k_j$

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } (j = i-1) \\ \min_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + w[i,j] \} & \text{if } (i \le j) \end{cases}$$

q_{i-1} → cost of searching (left subproblem) when $j = i-1$ or $i = j+1$

$e[i, r-1]$ → left subproblem

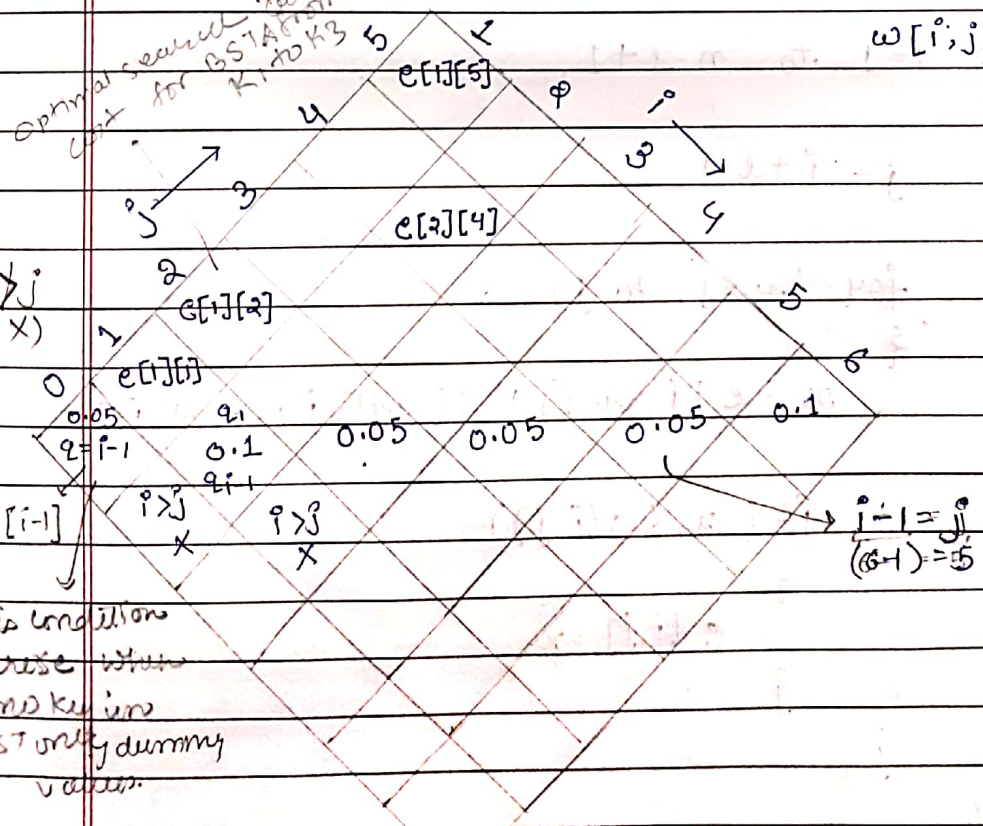
$e[r+1, j]$ → right subproblem

$$w[i,j] = w[i, r-1] + p_r + w[r+1, j]$$

p_r = probability of root

i	0	1	2	3	4	5
$k_i \leftarrow p_i$	–	0.15	0.1	0.05	0.1	0.2
$d_i \leftarrow q_i$	0.05	0.1	0.05	0.05	0.05	0.1

optimal search cost for BST created from k_1 to k_3



$w[i,j]$ = cost change due to inc. in height of BST as k_r is selected as root

For optimal solⁿ to BST we have to recursive eqⁿ based on two conditions -

Condition 1 → $j = i-1$ and we place the search cost as the probability of dummy key

20/July/2019

Step 3 : Algo to find optimal searching cost.

```
OBST ()
{
    c[6][6]    // if n=5
    for (i=1 to n+1)

        c[i][i-1] = q_i-1    // if j = i-1 (no key is BST only dummy values)

        for l=1 to n
        {
            for i=1 to n-l+1
            {
                j = i+l-1

                for (u=i to j)
                {
                    x = c[i, u-1] + c[u+1, j] + w[i,j]

                    if ( x < c[i,j])
                    {
                        c[i,j] = x
                    }
                }
            }
        }
}
```

if $i \leq j$

① $i == j$ → (BST has only 1 key)

② $i < j$ (>2 keys in BST)

$i=1 \Rightarrow i==j$ $i=2$ to $n \Rightarrow i<j$

Time Complexity $\simeq n^3$

* HEAP SORT : Heap (data structure)

① Algo
② definition
③ application
④ dry & run

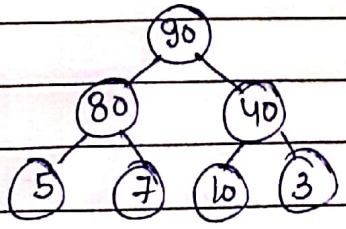
can be used to implement priority queue.

	0	1	2	3	4
element	96	18	70	11	81
Timestamp	8AM	9AM	10AM	12PM	2PM
Priority	4	2	1	5	2

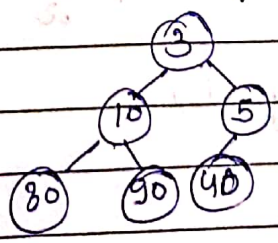
1 = highest priority

	0	1	2	3	4
Element	70	81	18	96	
Timestamp	10AM	2AM	9AM	8AM	
Priority	1	2	3	4	5

Max heap



Min heap.



- Two implementations of heap sort :-

(1) Heap sort (from unsorted array) $(n \log_2 n)$

(2) Restore heap (after deleting element from root) $(\log_2 n)$

20/Feb/2019

10	7	1	16	90	80
1	2	3	4	5	6

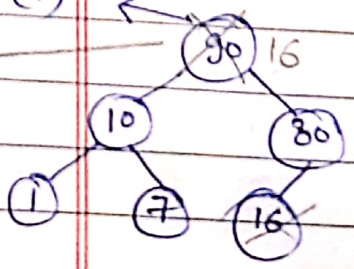
$n = 6$

$O(n)$ - to find max.

heap sort()

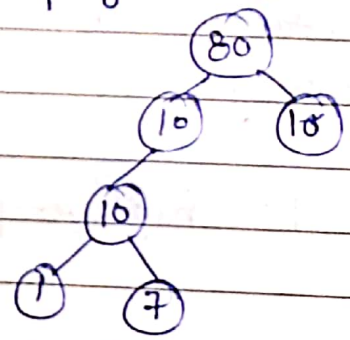
$O(1)$

90



(max)

Restore heap() == heapify $O(\log_2 n)$



Algorithm :-

```
Heapsort ()
{
  for i = n/2 to 1  // n/2
      Heapify (i)
}
```

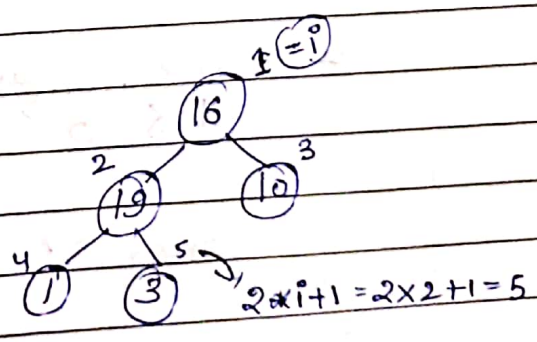
```
Heapify
{
  y = 2 * i
```

n = no. of elements in the array

```
int A[n];
Heapify(i)
{
    l = 2 * i
    r = 2 * i + 1
    if (l <= length(A) && A[l] > A[i])
        max = l
    else
        max = i

    if (r <= length(A) && A[r] > A[max])

        max = r

    if (max != i)
    {
        swap A[max] <--> A[i]

        Heapify(max)
    }
}
```

int A[n=5]

1	2	3	4	5
16	19	10	1	3



$n = 5$

$\frac{n}{2} = 3$ or 2

Complexity of Heapsort = no. of times Heapify called × Complexity of Heapify

$= \frac{n}{2} \times \log_2 n$

Dry Run:

Heapsort

$i = \frac{n}{2}$ to 1 $\Rightarrow \frac{5}{2} \leq 3$ to 1 $\quad i=3,\ i=2,\ i=1$

→ $i = 3 \qquad l = 2*i = 6 \qquad r = 2*i+1 = 7$

if $(\underset{6}{l} \not< \underset{5}{\text{length}}(A))$ $\qquad max = i = 3$

if $(\underset{7}{r} \not< \underset{5}{\text{length}}(A))$

if $(\underset{3}{max} != \underset{3}{i})$ False $\qquad$ (no swap() no Heapify call)

→ $i = 2 \qquad l = 2*i = 4 \qquad r = 2*i+1 = 5$

if $(\underset{4 \leq 5}{l \leq \text{length}(A)}$ && $\underset{1}{A[l]} > \underset{19}{A[i]})$

$max = i = 2$

if $(\underset{5 \leq 5}{r \leq \text{length}(A)}$ && $\underset{3}{A[r]} > \underset{19}{A[max]})$ False.

if $(\underset{2}{max} != \underset{2}{i})$ False $\qquad$ (no swap & no heapify call)

→ i = 1 $l = 2*i = 2$ $r = 2*i+1 = 3$.

if ($l \leq$ length(A)) && A[l] > A[i])
$2 \leq 5$ 19 > 16

max = $l = 2$

if (r ≤ length(A)) && A[r] > A[max])
3 ≤ 5   10   19

if (max != i) True, swap A[max] ↔ A[i]
2   1   2   1

Heapify (max)
↳ 2

- Heapsort function works on the array representation of tree
- To calculate the complexity of heap sort we multiply the no. of calls to Heapify with the complexity of Heapify. The complexity of Heapify will be $\log_2 n$ because Heapify fn has tail recursion and at the max it could be called for one complete branch of tree which is $\log_2 n$.
- Heap is complete Binary tree.

20	1	80	40	8	7	16	10

Create max heap from the given array.

27/Feb/2019

Date
DELTA Pg No.

* **Asymptotic Notation** : analyze time complexity

Analysis of Algorithm

- Asymptotic time complexity = no. of instructions executed by algo $\propto n$ (data size)
- Asymptotic Space Complexity = no. of memory spaces. $\propto n$

- Asymptotic space means not in bytes.
- Asymptotic time means not in milliseconds.

→ How to analyze algo -

(1) Manual analysis - looking at loops / recursion / function call.

(2) Recurrence Relation of algorithm to be solved.

$f(n) = 2n^2 + 1 \simeq n^2$ (= $g(n)$, growth function)

$f(n)$: calculated asymptotic time complexity.

→ Asymptotic Notation

(1) worst case — Big O, Small o.

Scanned by CamScanner

② Best case → Ω (big omega)
→ ω (small omega)

③ Avg case — θ (theta)

- Quick sort time complexity is n^2 → worst case $O(n^2)$
- Quick sort time complexity — $\Omega(n \log_2 n)$ ↓ best case

→ Algo 1

$f(n) = 2n^2 + 3n + 1 \simeq \underbrace{n^2}_{g(n)}$

Big O ⇒ $f(n) = O(g(n))$

$2n^2 + 3n + 1 = O(n^2)$

$f(n) = O(g(n))$ → if $0 < \overbrace{f(n)}^{2n^2+3n+1} \leq \underbrace{\text{constant}}_{c_1} \overbrace{g(n)}^{n^2}$

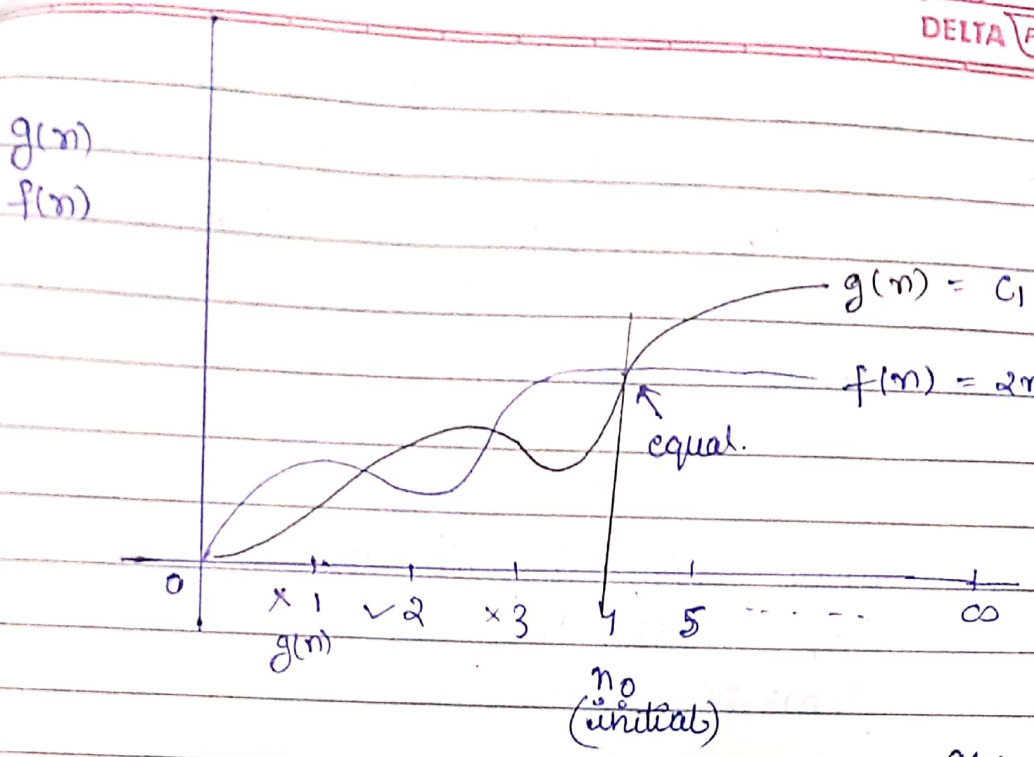
where $c_1 > 0$

$n \geq n_0 \geq 1$

worst case = upper bound

Big O and small o → $f(n)$ is bounded by $g(n)$ on upperside

$f(n) \leq g(n)$



→ Growth function [g(n)] a function in terms of data size, which shows how the no. of instructions grow inproportion to the growing data size for given algo, Growth functions are standared functions and relation b/w various growth functions is predefined.

eg — always $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2)$ ..... and so on.

- Big O notation - is Tight Upper Bound.
  Small o is Upper Bound.

* Small o :

$f(n) = o\,g(n)$

if $0 < f(n) < C_1 g(n)$

where $C_1 > 0$ and $n \geq n_0 \geq 1$

Any function $f(n)$ is said to be equal to $g(n)$ if the value of function $f(n)$ is always less than the values of $g(n)$ x constant $[C_1]$, for all size of data, which is greater than equal to initial data size n_0.

2 marks

Ques. Prove that $f(n) = O(g(n))$ where

$f(n) = 2n^2 + 3n + 1$

$g(n) = n^2$

Solu. $f(n) \leq C_1 g(n)$ for $C_1 = ?$ and $n_0 = ?$

To prove $\boxed{f(n) \leq g(n)}$

$$2n^2 + 3n + 1 \leq C_1 g(n)$$

$$= n^2 \left(2 + \frac{3}{n} + \frac{1}{n^2}\right) \leq C_1 (n^2) \quad [n=1]$$

$$= C_1 \geq 6$$

$\boxed{C_1 = 6}$

	$2n^2+3n+1$	$\leq 6n^2$	
$n=1$	$2+3+1$	$\leq 6*1$	T
$n=2$	$8+6+1$	$\leq 6(2^2)$	T
	Let $C_1=5$	$(2n^2+3n+1 \leq 5n^2)$	
$n=1$	$2+3+1$	$\leq 5(1)$	F ✗
$n=2$	$2(4)+3(2)+1$	$\leq 5(2)^2$	T

1st pair $C_1 \geq 6$ and $n \geq n_0$, $n_0 \geq 1$ $(C=6)$

2nd pair $C_1 = 5$ and $n_0 \geq 2$ $(C=5)$

| 20 | 1 | 80 | 40 | 8 | 7 | 16 | 10 |

Create max heap from the given array.

28/Feb/2019.

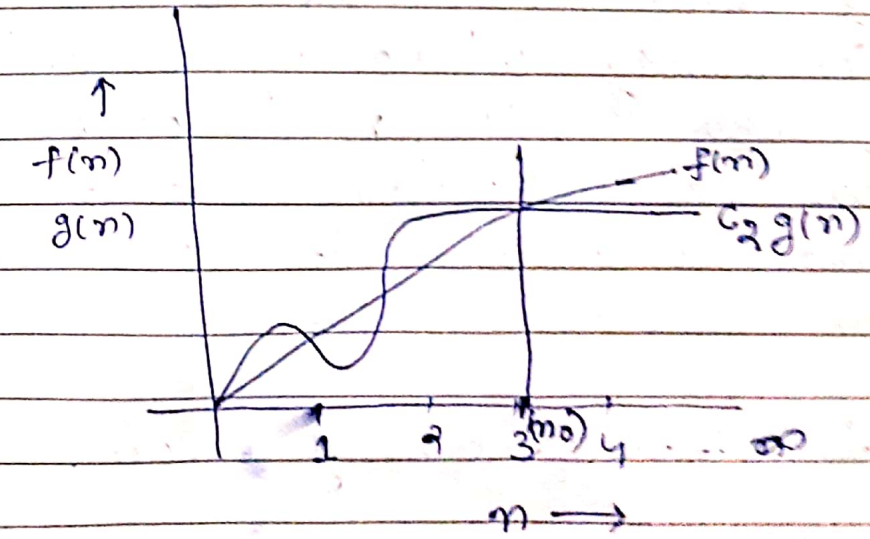
$g(n)$ - growth f^n or rate of growth of algo time.

① Worst case analysis — O (Big) (tight) — $g(n)$ - upper bound
                    — small (o) (loose) — upper bounds on algo complexity.

② Best case analysis — Ω (tight) — lower bound
                    — ω (loose) — to algo complexity.

$f(n) = \Omega\, g(n)$ if $0 < c_2 g(n) \leq f(n)$ (lower bound)
($n^2 \leq 2n^2 + 3n$)

$f(n) = \omega\, g(n)$ if $0 < c_2 g(n) < f(n)$



(Best case)

Ques. Prove that $f(n) = \Omega g(n)$ where $f(n) = 2n^2 + 3n + 1$

Ans. $f(n) = \Omega g(n)$ iff $0 < c_2 g(n) \leq f(n)$

$\frac{}{n^2} \quad \frac{}{2n^2+3n+1}$

$c_2 n^2 \leq 2n^2 + 3n + 1$

we have to calculate some (c_2 and n_0)

$\because \quad 2n^2 \leq 2n^2 + 3n + 1$

let select $c_2 = 2$, $n_0 \geq 1$ (pick the highest order terms)

$\because$ c_2 is selected as 2

	$c_2 n^2$ $= 2n^2$	$\leq 2n^2 + 3n + 1$
$n = 1$	$2(1)^2$	$\leq 2(1)^2 + 3(1) + 1$ ✓
$n = 2$	$2(2)^2$	$\leq 2(2)^2 + 3(2) + 1$ ✓

Other c_2 and n_0 pair.

let $c_2 = 3$

$n = 1$	$3(1)^2$	$\leq 2(1)^2 + 3(1) + 1$ ✓
$n = 2$	$3(2)^2$	$\leq 2(2)^2 + 3(2) + 1$ ✓
$n = 3$	$3(3)^2$	$\leq 2(3)^2 + 3(3) + 1$ ✓

③ theta (θ)
↑
average case analysis

we can calculate avg. case complexity of any algorithm only if the $f(n)$ of algorithm is bounded both at upper and lower end by same $g(n)$

$f(n) = \theta(g(n))$ iff $0 < c_2 g(n) \leq f(n) \leq c_1 g(n)$

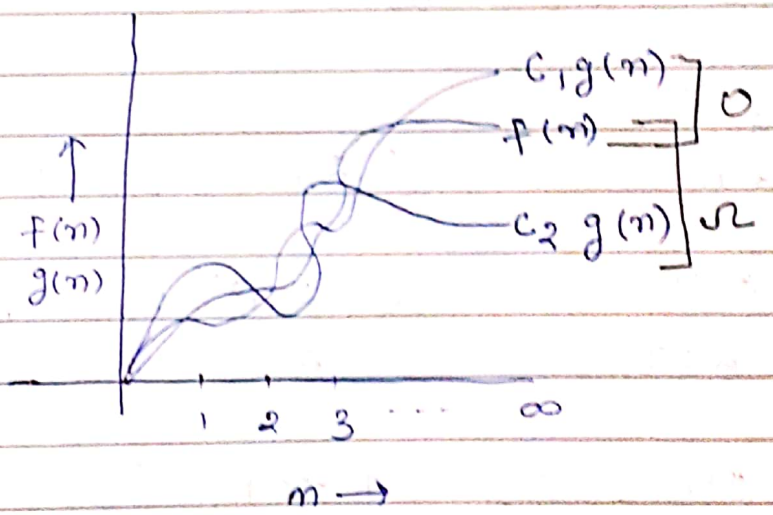
$f(n)$ ↓ calculated complexity of algo

Ω

$f(n)$ is bounded by same $g(n)$ at lower end.

bounded on upper end by $g(n)$

O (Big O)



Ques. Prove that $f(n) = 3n^2 + 3n + 1$ is $f(n) = \theta g(n)$

Ans. $\because f(n) = \theta g(n)$

$\Rightarrow 0 < c_2 g(n) \leq f(n) \leq c_1 g(n)$

and for $c_1 = 5$

and $n_0 \geq 2$

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

for $c_2 = 2$ and $n_0 \geq 1$

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

so,

$$0 < \underbrace{2n^2}_{c_2\, g(n)} \leq f(n) \leq \underbrace{5n^2}_{c_1\, g(n)}$$

- When we have both $g(n)$ value same, then only we can write,

$$\boxed{f(n) = \Theta\, g(n)}$$

Ques. If $f(n) = O\, g_1(n)$ ($g_1(n) = n^2$) for $c_1 = 2$ and $n_0 \geq 2$ and

$f(n) = \Omega\, g_2(n)$ ($g_2(n) = 10 \log n$) for $c_2 = 3$ and $n_0 \geq 1$

Is $f(n) = \Theta\, g(n)$. Justify (Y/N)

Ans. No, because $g_1(n) \neq g_2(n)$

if g_2 = not defined then also can't find Θ

or if g_1 = " " " " " "