Design And Analysis Of Algorithm notes part 2

Design And Analysis Of Algorithm (Islamic University of Science and Technology)

Heap:

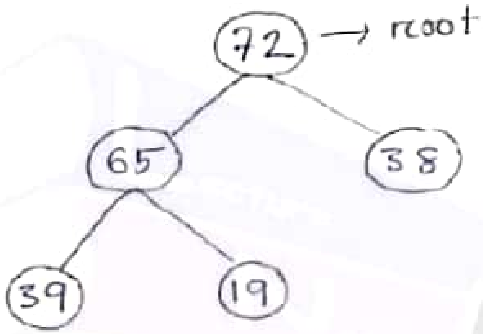
Heap is a complete binary tree where every parent is greater (or smaller) than children.

Heap is two types,

i) Max - heap
ii) Min - heap

Max-heap: Every parent is greater than children.

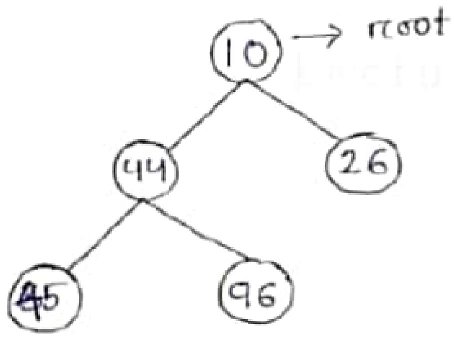
Example:



In max-heap the largest element is present at root.

Min-heap: Every parent is smaller than children.

Example:

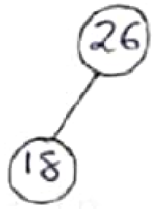


In min-heap the smallest element is present at root.

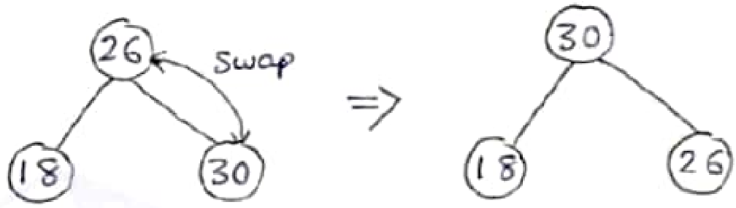
Q: Create max-heap with following elements.

26 18 30 32 16 74

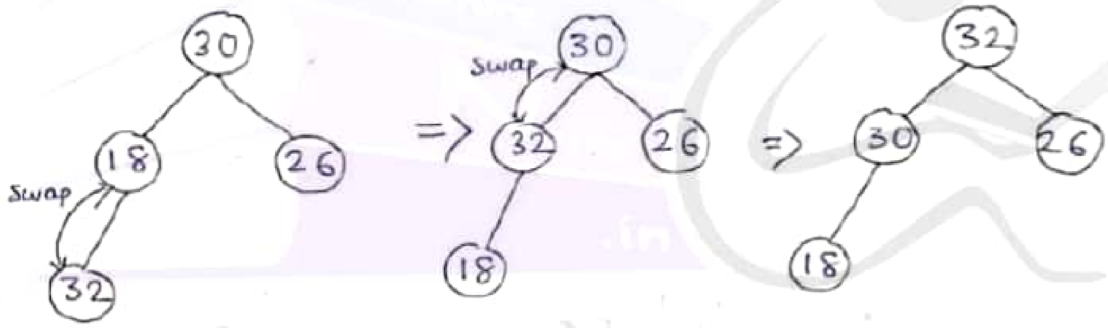
insert 26, 18



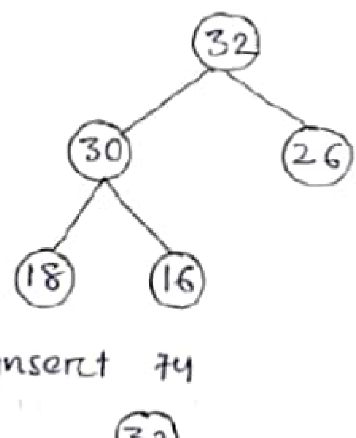
insert 30



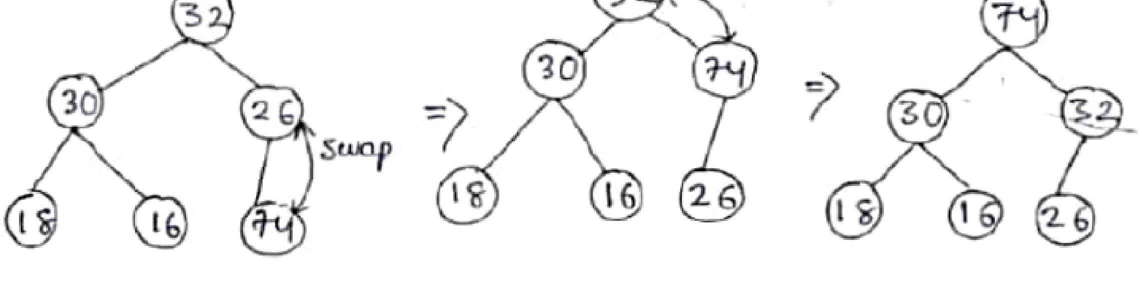
insert 32



insert 16

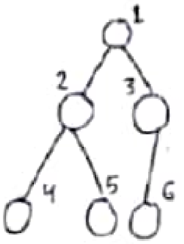


insert 74



Build maxheap: To build a maxheap, we should heapify all the parent nodes. Heapify means "maintain heap property i.e parent > children".

```
Buildmaxheap(n)
{
   // for all parent nodes
   for i = n/2 to 1
     heapify(i)
}
```



Here, $n=6 \Rightarrow \frac{n}{2} = 3$.
$i = \frac{n}{2}$ to $1 = 3$ to $1 = 3, 2, 1$
$\Rightarrow$ Parent nodes are 3, 2, 1. Parent node is the node having child.

heapify(i) makes parent > child at position i

If parent's position is i
Then. Leftchild's position is 2i
Rightchild's position is 2i+1

find Largest between Leftchild and right child.

If (Parent < largest) then swap the elements at parent and largest.

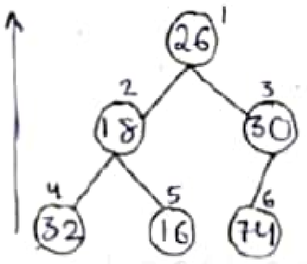
```
heapify(i)
{
    Parent = i
    Leftchild = 2i
    rightchild = 2i+1

    Largest = parent                                  check leftchild is present or not
    if(A[leftchild]>A[parent] && leftchild <= heapsize)
       Largest = leftchild
    if(A[rightchild]>A[parent] && rightchild <= heapsize)
       Largest = rightchild
    if(A[Parent] < A[largest])
       Swap (A[Parent], A[Largest])
    heapify (Largest)
       -> recursion
}
```

Example:

1	2	3	4	5	6
26	18	30	32	16	74

Build maxheap:

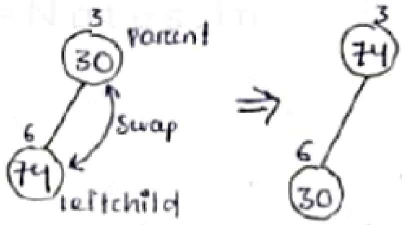


$n = 6$

$i = n/2$ to $1 = \frac{6}{2}$ to $1 = 3$ to $1 = 3, 2, 1$

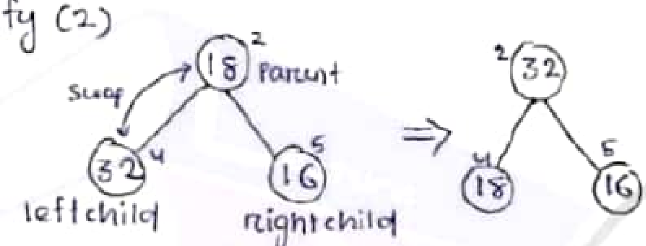
heapify(i) ⇒ heapify(3), heapify(2), heapify(1)

heapify (3)



$i = 3$
Parent = 3
leftchild = $2i = 6$
rightchild = not present

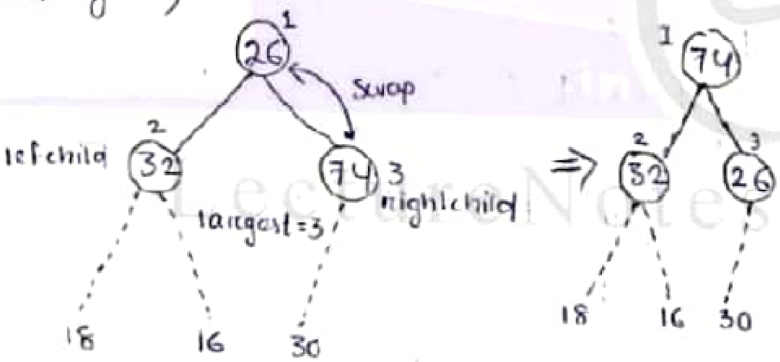
heapify (2)



largest = 4

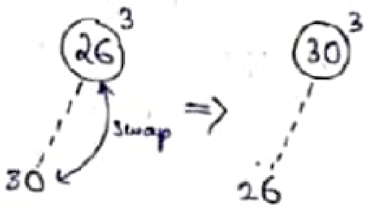
$i = 2$
Parent = 2
leftchild = $2i = 4$
rightchild = $2i+1 = 5$

heapify (1)

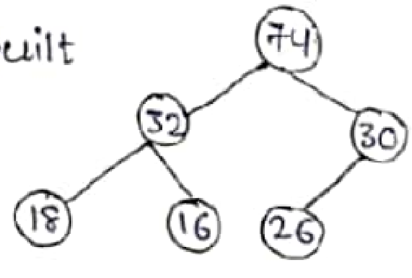


$i = 1$
Parent = 1
leftchild = $2i = 2$
rightchild = $2i+1 = 3$

Now, heapify (Largest)
⇒ heapify (3)



∴ maxheap is built



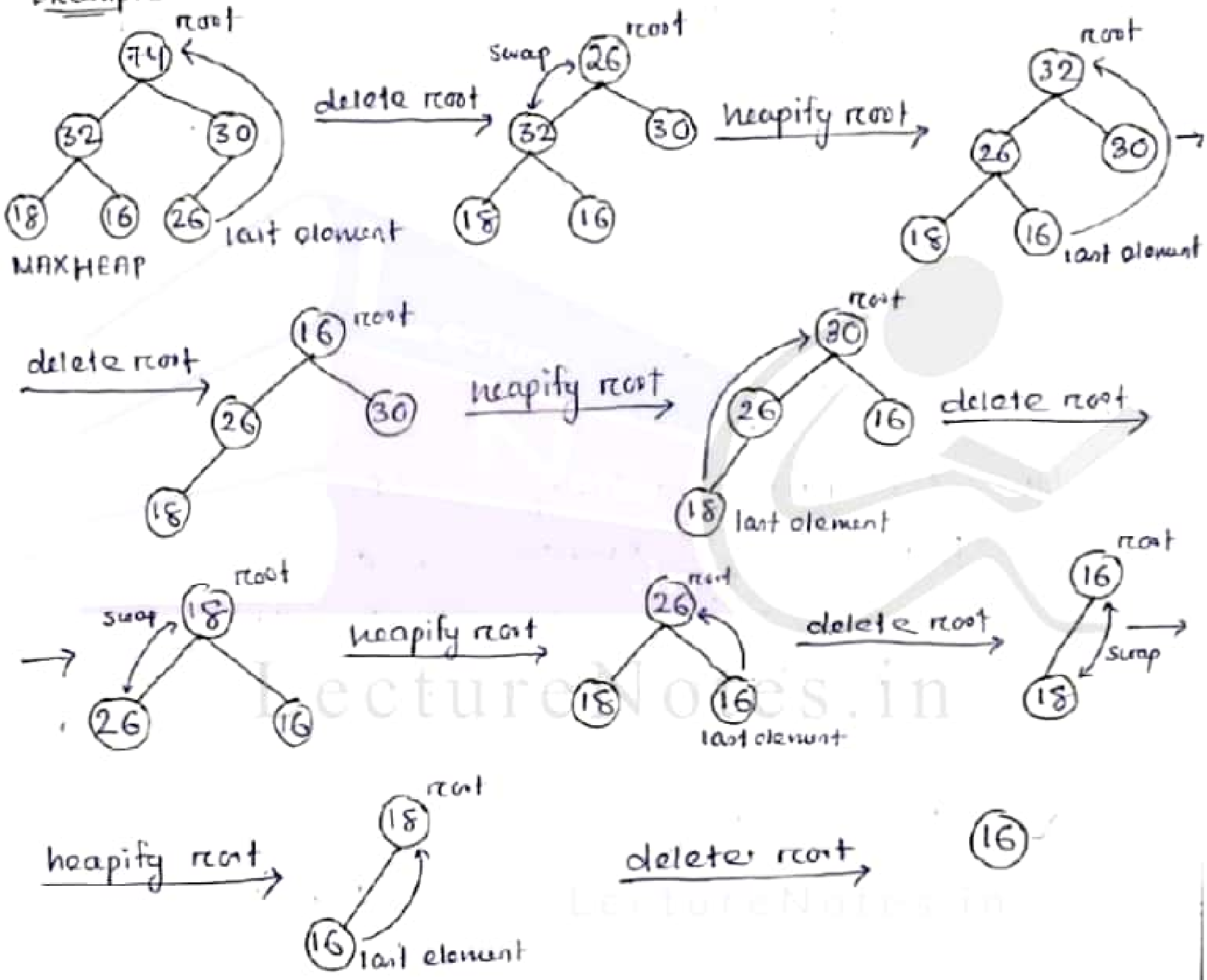
Heapsort : When we build a maxheap, the largest element is at root.

Delete root to get the Largest element. Again, make maxheap from remaining n-1 elements. Delete root to get the 2nd Largest element. So on.

Steps :

1. Build maxheap from n heapsize elements.
2. Delete the root.
3. heapify the new root.
4. Continue step2 and step3 till heapsize > 1

Example :



Deleted roots are stored at the last element of array → heap.

Stored array is.

16	18	26	30	32	74

Algorithm heapsort (A)

```
{
    // A is an array of n elements
      heap size = n
      Build maxheap(n)
    while (heapsize > 1)
    {
          root              last element
        swap (A[1] , A[heapsize])  } delete root
        heapsize = heapsize - 1    }
        heapify (1) ----------------> { heapify root
    }
}
```

Analysis of heapsort :

Time for Buildmaxheap = $O(n\log n)$

while loop executes $O(n)$ times,

Time for a heapify() = $O(\log n)$

=> Time for while loop $= O(n \times \log n)$

$= O(n\log n)$

Total time for heapsort $= O(n\log n + n\log n)$

$= O(2n\log n)$

$= O(n\log n)$

Lowerbound of sorting:

→ Lower bound means minimum time.
- For minimum time, we use Ω notation.

We know that,

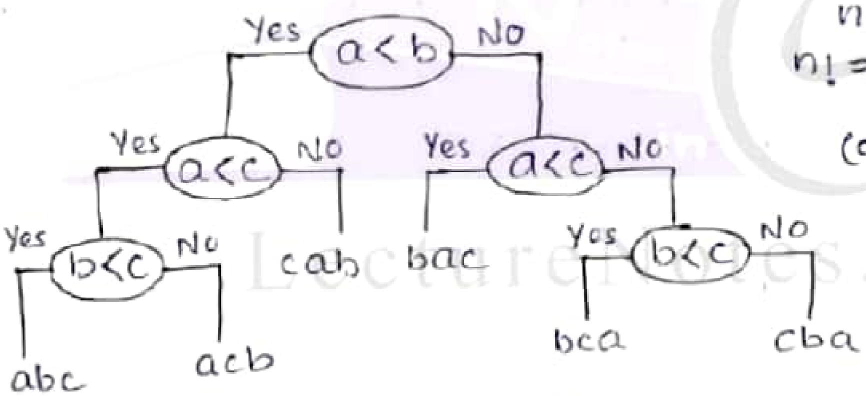
time taken by an algorithm = no. of comparisions.

- Comparision is also called binary comparision. That means comparision betn two elements.
- Merge sort & heap sort use binary comparision. These are called Comparision based sorting.
- Any sorting algorithm have atleast nlogn comparision

⇒ $\boxed{\text{Lower bound of comparision based sorting algorithm} = \Omega(n\log n)}$

Example:

Consider 3 elements a, b, c
the decission tree for a sorting algorithm is given below,



$n = 3$

$n! = 6$ number of leaf nodes.
(abc, acb, cab, bac, bca, cba)

Fig: Decission tree for comparision sorting

- A tree of height 'h' has maximum 2^h leaf nodes.
- It is found that total number leaf nodes $= n!$

that means, $2^h \geq n!$

$\Rightarrow \log 2^h \geq \log(n!)$

$\Rightarrow h\log 2 \geq \log(n!)$

$\Rightarrow h \geq \log n!$

$\Rightarrow h = \Omega(n\log n)$

$n! = n(n-1)(n-2)\cdots 1$

$n! = n^n\{1(1-\frac{1}{n})(1-\frac{2}{n})\cdots\frac{1}{n}\}$ (constant)

$n! \geq n^n$

$\log n! \geq \log n^n$

$\log n! \geq n\log n$

$\log n! = \Omega(n\log n)$

Here, h = height of decission tree
= no. of comparisions.

Priority queue :

- In priority queue, each element has a priority.
- In a priority queue, an element with high priority is served before an element with low priority.
- Applications : scheduling of jobs or programs.
- Priority queue is designed using heap.
- Priority queue is two types
  1. Max Priority queue.
  2. Min Priority queue.

→ Consider a max priority queue A

→ A has following operations,

1. Maximum (A) : Print maximum value.
2. Extractmax (A) : Extract maximum value.
   remove and return
3. Increasekey (A, i, Key) : Increase the key at Position i.
4. Insert (A, Key) : Insert the key in A
   Key means value.

Maximum - Priorityqueue

```
Algorithm maximum (A)
{
    Print A[1]   ↱ root
}
```

Extractmax - Priorityqueue

```
Algorithm Extractmax (A)
{
    max = A[1]          ↱ root
    A[1] = A[heapsize]  ↱ last element
    heapsize = heapsize - 1
    heapify (1);        ↱ root
    return max ;
}
```

IncreaseKey - Priorityqueue

Algorithm Increasekey (A, i, Key)

```
{
    if key < A [i]
    {
       error " Key is smaller than Current key "
    }
    A [i] ← Key
                   ↱root                  ↱child
    while (i > 1 and A [Parent] < A [i])
    {
        Swap ( A [Parent] ; A [i] )   { Move
          i = parent                  { to proper
                                      { Position.
    }
}
```

Insert - Priorityqueue

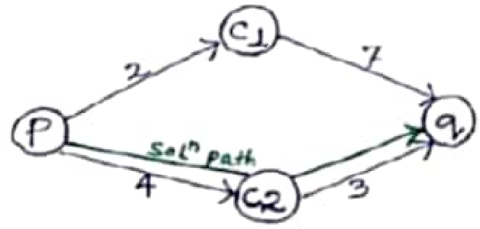
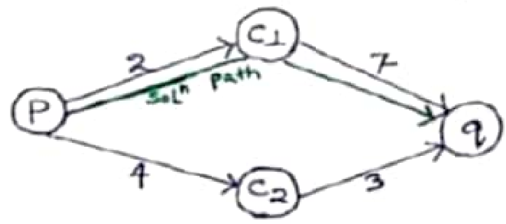
Algorithm Insert (A, Key)

```
{
      heapsize = heapsize + 1
                    ↱last element
      A [heapsize] = - ∞
      Increase Key (A, heapsize, Key)
}
```

Dynamic Programming

- Dynamic programming is similar to 'Divide And Conquer'
- It divides the problem into number of subproblems
- Solution of a subproblem is stored in a table for future use.

Dynamic programming	Divide And Conquer
• Subproblems are dependent • Same subproblem is not calculated every time it is required.	• Subproblems are independent • Same subproblem is calculated every time it is required.

Dynamic programming	Greedy
• It is an algorithm design technique	• It is an algorithm design technique
• Suitable for variety of problems	• Suitable for specific problems
• Many decission sequence is generated	• One decession sequence is generated
• Bottom up approach	• Top down approach
• Example: Longest Common Subsequence, Matrix chain multiplication	• Example: Activity selection problem, Assembly line scheduling, Fractional knapsack problem, Huffman Coding
Elements of dynamic programming 1. optimal substructure optimal solution of problem = combination of optimal solution of all subproblems [optimal solution = best solution] 2. Overlapping subproblems subproblems are solved again and again	**Elements of Greedy** 1. optimal substructure 2. Greedy property select the solution path which looks best right now.
Example: Shortest path from p to q 	Example: Shortest path from p to q 

Q: What is the similarity between Dynamic programming and Greedy

Ans: 1. use for optimization problem
2. Apply optimal substructure method

MCM (Matrix Chain Multiplication)

- Matrix chain means number of matrices
- We want to multiply number of matrices

Each matrix has an order i.e. row × column

Consider 2 matrices A_1, A_2

$$\underset{\substack{2\times 3\\ m\ \ n}}{\overset{A_1}{\begin{bmatrix} 3 & 6 & 4 \\ 5 & 1 & 7 \end{bmatrix}}} \underset{\substack{3\times 4\\ p\ \ q}}{\overset{A_2}{\begin{bmatrix} 2 & 6 & 5 & 8 \\ 6 & 7 & 4 & 5 \\ 9 & 3 & 2 & 4 \end{bmatrix}}} = \underset{\substack{2\times 4\\ m\ \ p}}{\overset{\text{Result-matrix}}{\begin{bmatrix} \quad & \quad \\ \quad & \quad \end{bmatrix}}}$$

Total no of Scalar multiplications $= m \times n \times q$
$= 2 \times 3 \times 4$
$= 24$

Example 1: Consider 3 matrices A_1, A_2, A_3

Order of $A_1 = 10 \times 7$
order of $A_2 = 7 \times 5$
order of $A_3 = 5 \times 20$

Consider the following cases of multiplication

case 1 : $A_1(A_2 A_3)$
case 2 : $(A_1 A_2) \cdot A_3$

Case 1 : $A_1(A_2 A_3)$

if A_2 & A_3 is multiplied
then no. of multiplications $= 7 \times 5 \times 20$
$= 700$
order of result matrix is 7×20
Now, multiply A_1 & Result matrix
No of multiplications $= 10 \times 7 \times 20$
$= 1400$
Total multiplications $= 700 + 1400$
$= 2100$

Case 2 : $(A_1 A_2) A_3$

if A_1 & A_2 is multiplied
then no of multiplications $= 10 \times 7 \times 5$
$= 350$
order of result matrix is 10×5
Now, multiply A_3 and Result matrix
No of multiplications $= 5 \times 20 \times 10$
$= 1000$
Total multiplications $= 350 + 1000$
$= 1350$

$\therefore$ case 2 is better than case 1, because it has less no. of multiplications.

Example 2 : consider 4 matrices A_1, A_2, A_3, A_4

consider following cases of multiplications

case 1 : $(((A_1 A_2) A_3) A_4)$

case 2 : $((A_1 (A_2 A_3)) A_4)$

case 3 : $(A_1 (A_2 (A_3 A_4)))$

case 4 : $((A_1 A_2) (A_3 A_4))$

⋮

bracket is called parenthesis.

bracket will decide the "sequence of multiplications"

sequence will decide the "total no of scalar multiplications"

The case having minimum no of multiplications is called best case (optimal case). So, it is called optimal parenthesization.

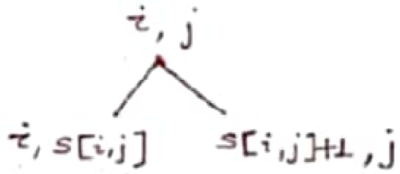
```
Algorithm Matrixchain(P)
{
    // P = number of orders
    // n = number of matrices
    for i = 1 to n
        m[i, i] = 0
          no of iterations
    for l = 2 to n
    {
        for i = 1 to n-l+1
        {
            j = i+l-1
            m[i, j] = ∞
            for k = i to j-1
            {
                q = m[i,k] + m[k+1,j] + P_{i-1} P_k P_j
                if q < m[i, j]
                {
                    m[i, j] = q     // m table stores multiplications
                    s[i, j] = k     // s table stores k value
                }
            }
        }
    }
}
```

Shortcut to Remember

```
l = 2 to n
{
    i = 1 to n-l+1
    {
        j = i+l-1
        k = i to j-1
    }
}
```

Note: M table and S table is of order n x n

```
Algorithm parenthesis( i, j )
{
    if ( i == j )
        Print A_i
    Else
    {
        Print (
        print parenthesis ( i, s[i,j] )
        Print parenthesis ( s[i,j]+1 , j )
        print )
    }
}
```



Q: Find optimal parenthesization to multiply following 3 matrices

$A_1(10\times 7)$, $A_2(7\times 5)$ $A_3(5\times 20)$

Ans: Order of matrices are $P_0 = 10$, $P_1 = 7$, $P_2 = 5$, $P_3 = 20$

$$m[i,j] = m[i,K] + m[K+1,j] + P_{i-1} P_K P_j$$

$$m[1,2] = \overset{0}{m[1,1]} + \overset{0}{m[2,2]} + \overset{10}{P_0}\cdot\overset{7}{P_1}\cdot\overset{5}{P_2} = 350$$

$$m[2,3] = \overset{0}{m[2,2]} + \overset{0}{m[3,3]} + \overset{7}{P_1}\cdot\overset{5}{P_2}\cdot\overset{20}{P_3} = 700$$

$$\left\{\begin{array}{l} m[1,3] = \overset{0}{m[1,1]} + \overset{700}{m[2,3]} + \overset{10}{P_0}\cdot\overset{7}{P_1}\cdot\overset{20}{P_3} = 2100 \\ m[1,3] = \overset{350}{m[1,2]} + \overset{0}{m[3,3]} + \overset{10}{P_0}\cdot\overset{5}{P_2}\cdot\overset{20}{P_3} = 1350 \end{array}\right\} \text{min}^n \text{ is } 1350$$

M Table

	1	2	3
1	0	350	~~2100~~ 1350
2		0	700
3			0

put zero in diagonal

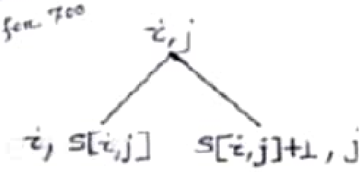
S Table

	1	2	3
1		1	2
2			2
3			

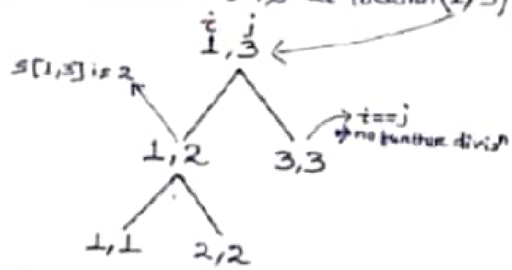
K value for 350

K value for 1350

K value for 700



Last value 1350 is at location (1,3)



bracket is given by looking above tree

$\Rightarrow ((A_1 A_2) A_3)$

Q: Find optimal parenthesization of matrix chain whose sequence of dimension is 4, 10, 3, 12, 20, 7.

Ans: Given, $P_0 = 4$, $P_1 = 10$, $P_2 = 3$, $P_3 = 12$, $P_4 = 20$, $P_5 = 7$

$$m[i,j] = m[i,K] + m[K+1,j] + P_{i-1} \cdot P_K \cdot P_j$$

$$m[1,2] = m[\overset{0}{1,1}] + m[\overset{0}{2,2}] + \overset{4}{P_0}\,\overset{10}{P_1}\,\overset{3}{P_2} = 120$$

$$m[2,3] = m[\overset{0}{2,2}] + m[\overset{0}{3,3}] + \overset{10}{P_1}\,\overset{3}{P_2}\,\overset{12}{P_3} = 360$$

$$m[3,4] = m[\overset{0}{3,3}] + m[\overset{0}{4,4}] + \overset{3}{P_2}\,\overset{12}{P_3}\,\overset{20}{P_4} = 720$$

$$m[4,5] = m[\overset{0}{4,4}] + m[\overset{0}{5,5}] + \overset{12}{P_3}\,\overset{20}{P_4}\,\overset{7}{P_5} = 1680$$

$$\left.\begin{aligned} m[1,3] &= m[\overset{0}{1,1}] + m[\overset{360}{2,3}] + \overset{4}{P_0}\,\overset{10}{P_1}\,\overset{12}{P_3} = 840 \\ m[1,3] &= m[\overset{120}{1,2}] + m[\overset{0}{3,3}] + \overset{4}{P_0}\,\overset{3}{P_2}\,\overset{12}{P_3} = 264 \end{aligned}\right\}\ \text{min}^m \text{ is } 264$$

$$\left.\begin{aligned} m[2,4] &= m[\overset{0}{2,2}] + m[\overset{720}{3,4}] + \overset{10}{P_1}\,\overset{3}{P_2}\,\overset{20}{P_4} = 1320 \\ m[2,4] &= m[\overset{360}{2,3}] + m[\overset{0}{4,4}] + \overset{10}{P_1}\,\overset{12}{P_3}\,\overset{20}{P_4} = 2760 \end{aligned}\right\}\ \text{min}^m \text{ is } 1320$$

$$\left.\begin{aligned} m[3,5] &= m[\overset{0}{3,3}] + m[\overset{1680}{4,5}] + \overset{3}{P_2}\,\overset{12}{P_3}\,\overset{7}{P_5} = 1932 \\ m[3,5] &= m[\overset{720}{3,4}] + m[\overset{0}{5,5}] + \overset{3}{P_2}\,\overset{20}{P_4}\,\overset{7}{P_5} = 1140 \end{aligned}\right\}\ \text{min}^m \text{ is } 1140$$

$$\left.\begin{aligned} m[1,4] &= m[\overset{0}{1,1}] + m[\overset{1320}{2,4}] + \overset{4}{P_0}\,\overset{10}{P_1}\,\overset{20}{P_4} = 2120 \\ m[1,4] &= m[\overset{120}{1,2}] + m[\overset{720}{3,4}] + \overset{4}{P_0}\,\overset{3}{P_2}\,\overset{20}{P_4} = 1080 \\ m[1,4] &= m[\overset{264}{1,3}] + m[\overset{0}{4,4}] + \overset{4}{P_0}\,\overset{12}{P_3}\,\overset{20}{P_4} = 1224 \end{aligned}\right\}\ \text{min}^m \text{ is } 1080$$

$$\left.\begin{aligned} m[2,5] &= m[\overset{0}{2,2}] + m[\overset{1140}{3,5}] + \overset{10}{P_1}\,\overset{3}{P_2}\,\overset{7}{P_5} = 1350 \\ m[2,5] &= m[\overset{240}{2,3}] + m[\overset{1680}{4,5}] + \overset{10}{P_1}\,\overset{12}{P_3}\,\overset{7}{P_5} = 2760 \\ m[2,5] &= m[\overset{1320}{2,4}] + m[\overset{0}{5,5}] + \overset{10}{P_1}\,\overset{20}{P_4}\,\overset{7}{P_5} = 2720 \end{aligned}\right\}\ \text{min}^m \text{ is } 1350$$

$$\left.\begin{aligned} m[1,5] &= m[\overset{0}{1,1}] + m[\overset{1350}{2,5}] + \overset{4}{P_0}\,\overset{10}{P_1}\,\overset{7}{P_5} = 1630 \\ m[1,5] &= m[\overset{120}{1,2}] + m[\overset{1140}{3,5}] + \overset{4}{P_0}\,\overset{3}{P_2}\,\overset{7}{P_5} = 1344 \\ m[1,5] &= m[\overset{264}{1,3}] + m[\overset{1680}{4,5}] + \overset{4}{P_0}\,\overset{12}{P_3}\,\overset{7}{P_5} = 2280 \\ m[1,5] &= m[\overset{1080}{1,4}] + m[\overset{0}{5,5}] + \overset{4}{P_0}\,\overset{20}{P_4}\,\overset{7}{P_5} = 1640 \end{aligned}\right\}\ \text{min}^m \text{ is } 1344$$

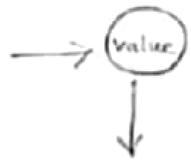
M table and S Table is of order 5×5

j →

M Table

i ↓

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

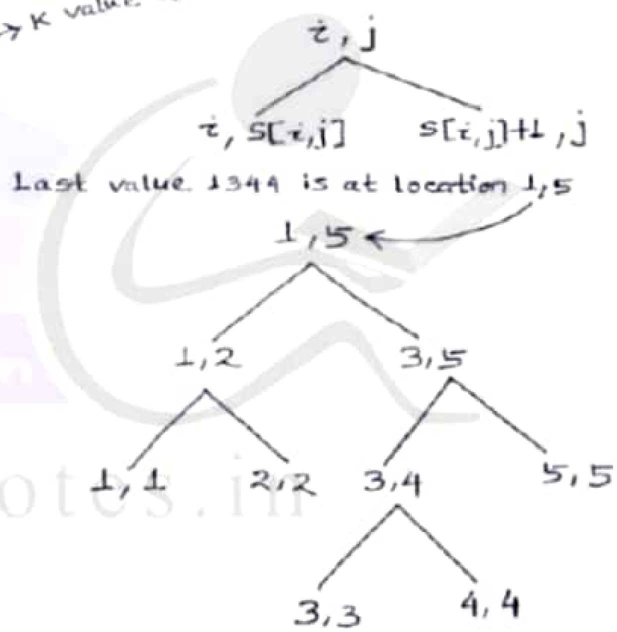


S Table

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

→ K value of 1080

→ K value of 1344



Parenthesis is given by looking above tree

$\Rightarrow ((A_1 A_2)((A_3 A_4) A_5))$

$$\boxed{\text{Time complexity of MCM} = O(n^3)}$$

LCS (Longest Common Subsequence)

Sequence means string (or set of characters)

Subsequence means substring (or part of a string)

LCS = common subsequence bet^n 2 strings whose length is longest.

Applications : File comparison, DNA comparison

Example: consider two strings X and Y

X = sagen	X = deepankar	X = sarmistha
Y = swadhin	Y = dipti	Y = sobha
LCS is san	LCS is dp	LCS is sha

X = BBACA

Y = BCBA

Common sequence = {BB, BA, BBA, BC, CA, BCA}

⇒ LCS = {BBA, BCA}

Q: How to create LCS Table

Ans: 1. No of rows in table = $m+1$ (m → length of X)

No of columns in table = $n+1$ (n → length of Y)

2. Fillup the First row (row0) and First column (column0) with Zero

3. Write x_i from row1 onwards

Write y_j from column1 onwards

4. Compare row$[x_i]$ with column$[y_j]$ as follows

Case1 : **EQUAL** $[x_i == y_j]$

Store (diagonal value + 1) and ↖

Case2 : **NOT EQUAL** $[x_i \neq y_j]$

Find greater bet^n top and Left

(a) if top > Left then store top value and ↑

if top == left " " "

(b) if left > top then store left value and ←

Q: Find LCS of X = ABACA and Y = BCBBA

$Y_j \longrightarrow$

$X_i \downarrow$

	column 0	B	C	B	B	A
row 0	0	0	0	0	0	0
A	0	0↑	0↑	0↑	0↑	1↖
B	0	1↖	1←	1↖	1↖	1↑
A	0	1↑	1↑	1↑	1↑	2↖
C	0	1↑	2↖	2←	2←	2↑
A	0	1↑	2↑	2↑	2↑	3↖

LCS is BCA

Q: Find LCS of X = ABCBDAB and Y = BDCABA

$Y_j \longrightarrow$

$X_i \downarrow$

	column 0	B	D	C	A	B	A
row 0	0	0	0	0	0	0	0
A	0	0↑	0↑	0↑	1↖	1←	1↖
B	0	1↖	1←	1←	1↑	2↖	2←
C	0	1↑	1↑	2↖	2←	2↑	2↑
B	0	1↖	1↑	2↑	2↑	3↖	3←
D	0	1↑	2↖	2↑	2↑	3↑	3↑
A	0	1↑	2↑	2↑	3↖	3↑	4↖
B	0	1↖	2↑	2↑	3↑	4↖	4↑

LCS is BCBA

Note: If we write X_i in column & Y_j in row then also we get correct answer.

```
Algorithm LCS(x, y)
{
  // m = length of x
  // n = length of y
  // C table store values
  // B table store symbols (↖, ←, ↑)

    for i = 0 to m
        c[i,0] = 0                       { Fill up First row with Zero
    for j = 0 to n
        c[0,j] = 0                       { Fill up First column with Zero

    for i = 1 to m                       { m rows
      for j = 1 to n                     { n columns
          if x_i == y_j
          {
  EQUAL         c[i,j] = c[i-1,j-1] + 1
                b[i,j] = ↖                   { Diagonal
          }
          else
          {                  →Top       →Left
             if c[i-1,j] ≥ c[i,j-1]
             {
  NOT EQUAL      c[i,j] = c[i-1,j]
                 b[i,j] = ↑                  { Top
             }
             else
             {
                 c[i,j] = c[i,j-1]           { Left
                 b[i,j] = ←
             }
          }
}
```

$j \rightarrow$

$i \downarrow$

i-1, j-1 Diagonal	i-1, j Top
i, j-1 Left	i, j

row i-1 → TOP

row i

column j-1 → left | column j

$\boxed{\text{Time complexity of LCS} = O(mn)}$

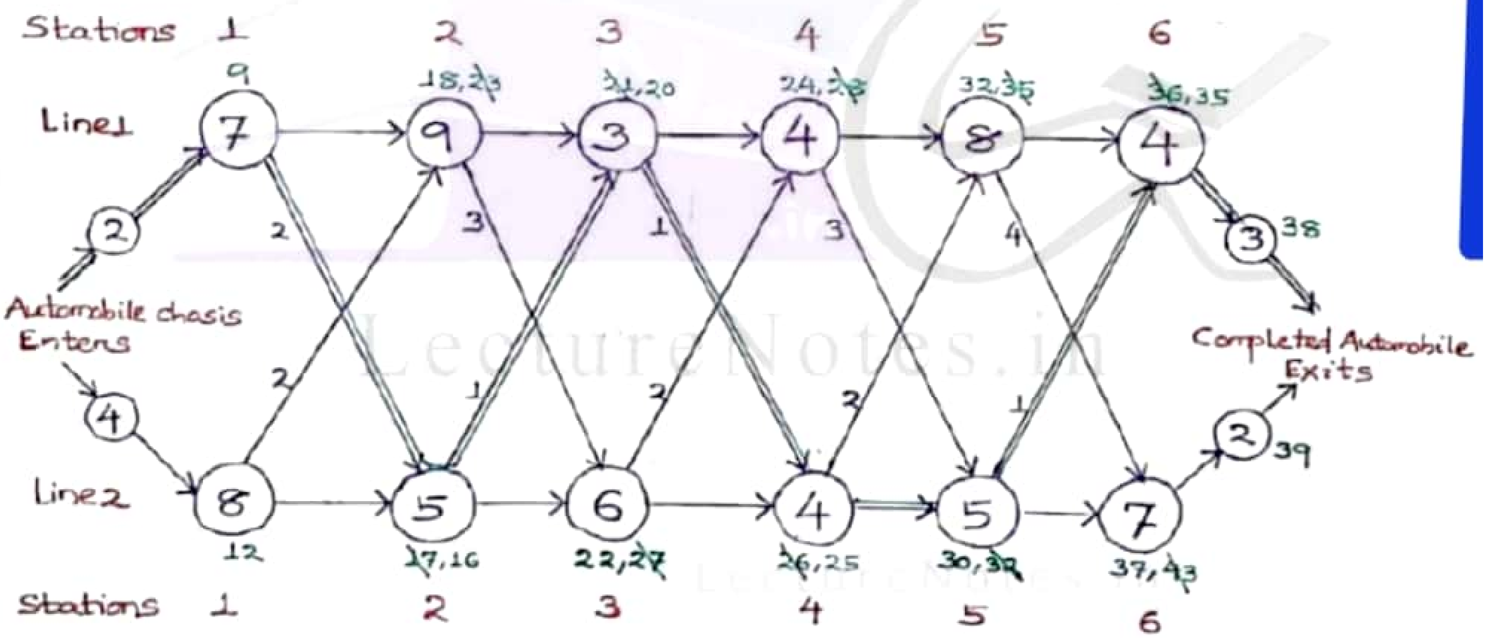
Assembly line scheduling

- An automobile company has two assembly lines
- An assembly line has n number of stations
- At each station some parts are assembled. The time to assemble is called assembly time.
- Automobile chasis enter at each assembly line
  The completed automobile exits at the end
- We will find "minimum time to assemble the automobile"

Example: Number of stations = n = 6

At each station, two values are written
1. Direct path from same line
2. Indirect path from different line

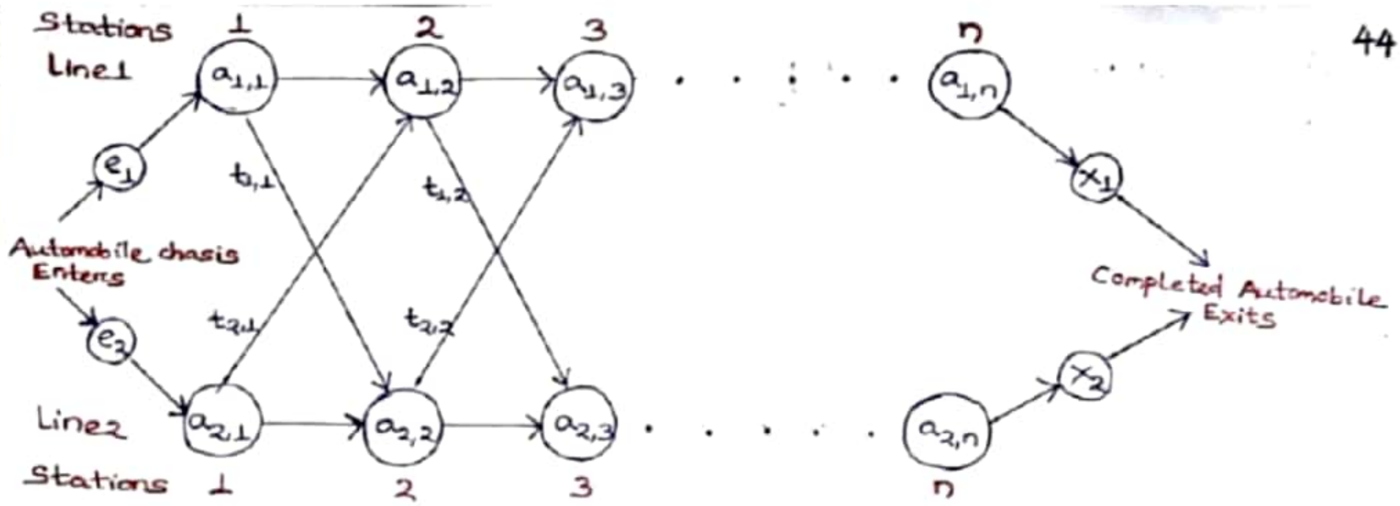
Greater value is crossed (not taken)



Minimum Time = 38

Green line shows the minimum path 38

This line is drawn by moving backward from 38



$a_{i,j}$ = assembly time $\quad$ $t_{i,j}$ = time for one station to another

i = assembly line = 1, 2 $\quad$ j = station number = 1, 2, 3, ... n

e_1 = time to enter line1 $\quad$ e_2 = time to enter line2

x_1 = time to exit line1 $\quad$ x_2 = time to exit line2

Algorithm Assemblyline (a, t, e, x, n)

$f_1[1] = e_1 + a_{1,1}$

$f_2[1] = e_2 + a_{2,1}$

for $j = 2$ to n

{

if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ (previous station of line1; previous station of line2)

$f_1[j] = f_1[j-1] + a_{1,j}$

$l_1[j] = 1$

Else

$f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$

$l_1[j] = 2$

} Line1

if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$

$f_2[j] = f_2[j-1] + a_{2,j}$

$l_2[j] = 2$

Else

$f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}$

$l_2[j] = 1$

} Line2

}

if $f_1[n] + x_1 \leq f_2[n] + x_2$

$f^* = f_1[n] + x_1$ // f^* stores min^m time

$l^* = 1$ // l^* stores line no at min^m time

Else

$f^* = f_2[n] + x_2$

$l^* = 2$

Algorithm printstations

$i = l^*$

Print line i station n

for $j = n$ to 2

{ $i = l_i[j]$, Print line i station $j-1$ }

Activity Selection problem

- Activity means task or event. Example: Seminar, meeting etc
- Every activity i has start time S_i, finish time f_i
- Activities are to be organized in a common Room (resource).
- Activities must be compatible. Compatible means 'no overlap'
- A_i, A_j are compatible $\Rightarrow$ start time of $A_j \geq$ Finish time of A_i
- We will find "Maximum number of compatible Activities"

Steps:

1. Arrange (sort) activities in increasing order of finish time
2. Select first Activity
3. Select next activity whose start time $\geq$ finish time of previously selected Activity.
4. Repeat step3 till all activities are checked.

Example1: Assume that activities are arranged in increasing order of finish time (as shown below)

	✓	✗	✓	✓	✗	✓
Activity A_i	1	2	3	4	5	6
Start time S_i	2	1	5	7	10	14
Finish time f_i	3	4	6	11	12	17

Select A_1
A_1 is finished at 3
So next Activity must start ≥ 3
But A_2 start at 1. So, A_2 is not taken
A_3' start at 5. So, A_3 is taken
So on...

Max^m compatible activities are $\{A_1, A_3, A_4, A_6\}$.

Max^m number of compatible activities $= 4$

Algorithm Activityselection(A)

{

// A stores 'n' activities arranged as per finish time

// SA stores selected Activity

SA = A_1 // Select the first activity A_1

Previous = 1

For $i = 2$ to n

Activity to select

Previous Activity

if start-time(A_i) $\geq$ finish-time ($A_{Previous}$)

{

union

SA = SA $\cup$ A_i // Select Activity A_i

Previous = i

}

}

Example 2:

Activity A_i	1 ✓	2 ✗	3 ✗	4 ✓	5 ✗	6 ✗
Starttime S_i	1	3	0	5	3	5
Finish time f_i	4	5	6	7	8	9

Max^m compatible activities are { A_1, A_4 }

Max^m no of compatible activities = 2

Example3:

Activity A_i	1 ✓	2 ✗	3 ✗	4 ✓	5 ✗	6 ✗	7 ✗	8 ✓	9 ✗	10 ✗	11 ✓
Start time S_i	1	3	0	5	3	5	6	8	8	2	12
Finish time f_i	4	5	6	7	8	9	10	11	12	13	14

Max^m compatible activities = {A_1, A_4, A_8, A_{11}}

Max^m no.of compatible activities = 4

Time complexity = $O(\underbrace{n\log n}_{\text{sorting}}) + O(\underbrace{n}_{\text{compare n activity}}) = O(n\log n)$

Knapsack problem

- Knapsack means bag
- Bag is to be filled with different items
- Each item has a weight and profit (cost)
- Weight or capacity of knapsack = W
  e.g. weight of knapsack = 15kg
- Fill the knapsack with items so that profit is max^m

Knapsack problem is 2 types

1. Fractional Knapsack : we can take fraction of an item
2. 0/1 knapsack : we can't take fraction of an item
   - → take total item
   - → donot take item

Note : Fractional knapsack is greedy algorithm
0/1 knapsack is dynamic programming algorithm.

Fractional Knapsack problem

steps

1. Arrange items in decreasing order of "profit by weight" ($\frac{P_i}{W_i}$)
2. put items one by one until the knapsack is full

If item taken $= x_i$ then profit $= P_i x_i$
weight $= W_i x_i$

we want to maximize profit but condition is $W_i x_i \leq W$

Mathematically

$$\boxed{\begin{aligned} &\text{Maximize } \sum_{i=1}^{n} P_i x_i \\ &\text{Condition : } \sum_{i=1}^{n} W_i x_i \leq W \end{aligned}}$$

no of items — size of knapsack

Example1: $n = 3$, $W = 14$

$$P_1, P_2, P_3 = 90, 100, 50$$

$$w_1, w_2, w_3 = 6, 8, 3$$

items	P_i (Profit)	w_i (weight)	$\frac{P_i}{w_i}$
item1	Rs 90	6 kg	$\frac{90}{6} = 15$
item2	Rs 100	8 kg	$\frac{100}{8} = 12.5$
item3	Rs 50	3 kg	$\frac{50}{3} = 16.6$

Arrange items in decreasing order of $\frac{P_i}{w_i}$
= item 3, item 1, item 2

$$\boxed{\begin{array}{ll} w_i \le W \Rightarrow x_i = 1 & , W = W - w_i \\ w_i \not\le W \Rightarrow x_i = \frac{W}{w_i} & , W = 0 \end{array}}$$

items	$w_i \le W \Rightarrow$	x_i	W
item 3	$3 \le 14 \Rightarrow$	1	$14 - 3 = 11$
item 1	$6 \le 11 \Rightarrow$	1	$11 - 6 = 5$
item 2	$8 \not\le 5 \Rightarrow$	$\frac{5}{8} = .6$	0

Algorithm Fractional knapsack (P, w, x)

```
{
    // P store profits, w stores weights, x stores fractional item
    // Arrange items in decreasing order of Pi/wi
    for i = 1 to n
    {
        if wi <= W
        {
            xi = 1              // Full item is taken
            W = W - wi
        }
        Else
        {
            xi = W/wi           // Fractional item is taken
        }   W = 0
    }
}
```

Example2: $n=5$, $W=9$, $P_1, P_2, P_3, P_4, P_5 = 10, 5, 6, 18, 3$

$w_1, w_2, w_3, w_4, w_5 = 2, 3, 1, 4, 1$

items	P_i (Profit)	w_i (weight)	$\frac{P_i}{w_i}$
item1	10	2	$\frac{10}{2} = 5$
item2	5	3	$\frac{5}{3} = 1.6$
item3	6	1	$\frac{6}{1} = 6$
item 4	18	4	$\frac{18}{4} = 4.5$
item 5	3	1	$\frac{3}{1} = 3$

Arrange items in decreasing order of $\frac{P_i}{w_i}$

= item3, item1, item4, item5, item2

items	$w_i \le W$	x_i	W
item 3	$1 \le 9$	1	$9-1=8$
item1	$2 \le 8$	1	$8-2=6$
item4	$4 \le 6$	1	$6-4=2$
item 5	$1 \le 2$	1	$2-1=1$
item2	$3 \not\le 1$	$\frac{1}{3}=0.3$	0

Example3: $n=3$, $W=20$, $P_1, P_2, P_3 = 25, 24, 14$

$w_1, w_2, w_3 = 18, 15, 10$

items	P_i	w_i	$\frac{P_i}{w_i}$
item1	25	18	$\frac{25}{18} = 1.3$
item 2	24	15	$\frac{24}{15} = 1.6$
item3	14	10	$\frac{14}{10} = 1.4$

Arrange items in decreasing order of $\frac{P_i}{w_i}$ = item2, item3, item1

items	$w_i \le W$	x_i	W
item2	$15 \le 20$	1	$20-15=5$
item 3	$10 \not\le 5$	$\frac{5}{10}=0.5$	0
item1	$18 \not\le 0$	$\frac{0}{18}=0$	0

Time complexity of fractional knapsack $= O(n\log n)$

Huffman Coding

- Data is stored in memory as bits. One bit is either 0 or 1.
- Coding means assigning →giving a code for the data.
- Coding stores (or represents) data in a specific format.
- Advantages of coding : Data compression, Data Security

  Data compression – Coding decreases no. of bits for data. Hence, less memory is needed for data.

  Data Security – Coding provides data security.

- Two types of coding

  1. Fixed Length coding : Each code has fixed number of bits
  2. Variable Length coding : Each code has variable number of bits

- Huffman coding is a variable length coding.

Steps:

step1 : Arrange →sort characters in increasing order of frequency

step2 : Create binary tree as following

Leftchild = 1st minimum frequency

rightchild = 2nd " "

Parent node = left child + rightchild

step3 : Add the parent node to list of elements.

step4 : Repeat step1 to step3 n-1 times

step5 : Assign every left branch with 0

Assign every right branch with 1

Algorithm Huffman(A)

{

// A is a table which stores characters and frequency

// n = no. of characters

for $i = 1$ to $n-1$

{

Sorting (A)

find min^m

Leftchild = extractmin(A) // 1st minimum

Rightchild = extractmin(A) // 2nd minimum

Parent = leftchild + Rightchild

Add parent to A

}

}

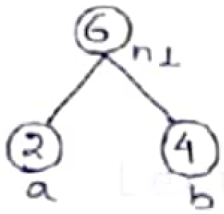
Time complexity of Huffman Algorithm $= O(n \log n)$

Example 1:

Character	Frequency
a	2
b	7
c	4

Arrange in increasing order: 2 4 7
a c b

Leftchild = 2
rightchild = 4
Parent = 2+4 = 6
Consider parent name is n1

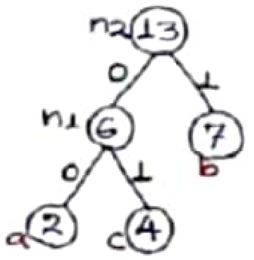


Add n1 to the list of elements : ~~2~~ ~~4~~ 7 6
~~a~~ ~~c~~ b n1

neglect the elements which are completed.

Arrange in increasing order : 6 7
n1 b

leftchild = 6
rightchild = 7
parent = 6+7 = 13
consider parent name is n2

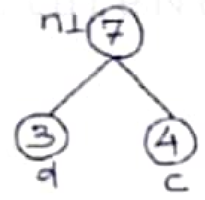


code of a = 00
code of b = 1
code of c = 01

Example 2:

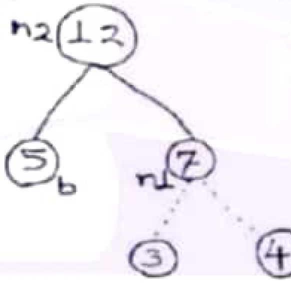
Character	Frequency
a	9
b	5
c	4
d	3

Arrange in increasing order : 3 4 5 9
d c b a



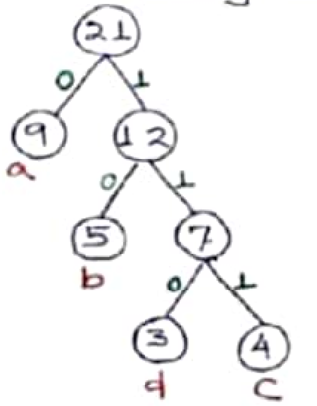
Add n1 : 5 9 7
b a n1

Arrange in increasing order : 5 7 9
b n1 a



Add n2 : 9 12
a n2

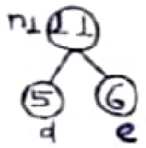
Arrange in increasing order : 9 12
a n2



code of a = 0
code of b = 10
code of c = 111
code of d = 110

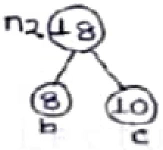
Example3: a-15, b-8, c-10, d-5, e-6

Arrange in increasing order: 5 6 8 10 15
d e b c a



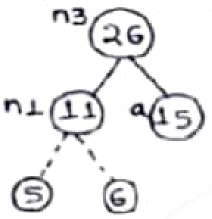
Add n1: 8 10 15 11
b c a n1

Arrange in increasing order: 8 10 11 15
b c n1 a



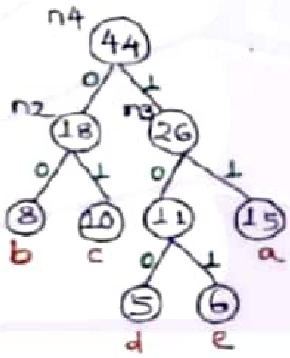
Add n2: 11 15 18
n1 a n2

Arrange in increasing order: 11 15 18
n1 a n2



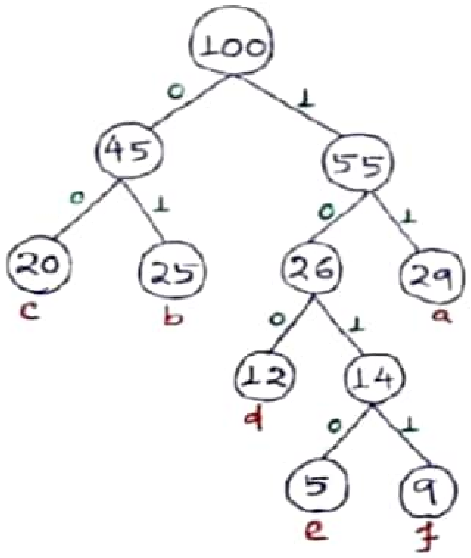
Add n3: 18 26
n2 n3

Arrange in increasing order: 18 26
n2 n3



code of a = 11
code of b = 00
code of c = 01
code of d = 100
code of e = 101

Example4: a-29, b-25, c-20, d-12, e-5, f-9



code of a = 11
code of b = 01
code of c = 00
code of d = 100
code of e = 1010
code of f = 1011

$\boxed{\text{Time complexity of Huffman Coding} = O(n\log n)}$