* Dynamic Programming
- Dynamic Programming is an optimization technique used to solve problem by breaking them down into overlapping Subproblems and storing their solutions to avoid redundant computations.
- It is especially useful for problems that exhibit optimal substructure (the solution to a problem depends on solutions to its subproblems) and overlapping subproblems (the same subproblems are solved multiple times

* Key Concepts of Dynamic Programming.
1. Memoization (Top-Down Approach)
   - Solve problems recursively.
   - Store results of subproblems in a table (usually an array or dictionary) to avoid recomputation.

2. Tabulation (Bottom-Up Approach).
   - Build a table iteratively from smaller subproblems to larger ones.
   - Avoid recursion overhead and often improves efficiency.

* Exploding Problem: It is a case where a recursion leads to exponential time complexity due to redundant subproblem calculations.
   - Polynomial Time (n^c).

* Matrix Chain Multiplication Problem:

- The matrix chain multiplication is a classic example of Dynamic programming.
- The goal is to determine the most effective / efficient way to multiply a given sequence of matrices while minimizing the number of scalar multiplications.
- Given n matrices $A_1, A_2 \ldots A_n$ where A_i has dimensions $p_{i-1} \times p_i$ the objective is to find the most efficient way to parenthesize the matrices to minimize the number of scalar multiplications.

Let dp[i][j] represent the minimum number of scalar multiplications required to multiply from A_i to A_j

The recurrence relation is.

$$dp[i][j] = \min_{i \le k < j} \left(dp[i][k] + dp[k+1][j] + p_{i-1} \times p_k \times p_j \right)$$

- where k represents the index at which multiplication is split.
- dp[i][k] computes the cost for multiplying matrices from A_i to A_k.
- $p[i-1] \times p[k] \times p[j]$ represents the cost of multiplying the two resulting matrices

For example matrices (A_1, A_2, A_3, A_4) can be fully parenthesized in five ways.

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

Total No of Possible ways =

$$T(n) = \frac{2nCn}{n+1}$$

Function M(i,j)

```
if i=j then return M=0;
else min = +∞ (very high value)
    for k = i to j-1 do
        mult = M(i,k) + M(k+1,j) + P[i-1] * P[k] * P[j];
        if mult < Min then
            min = mult;
        end if.
    end for.
end if.
```

This algorithm is able to find minimum number of multiplications using recursive approach.

- For every k, we make two subproblems
  a) Chain from 1 to k
  b) Chain from k+1 to j
- each subproblem can be further partitioned into smaller subproblems and we can find the total required multiplications by solving for each of the groups.

• The base case would be when we have only one matrix left which means $i = j$.

Complexity

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k)) + O(n)$$

$T(1) = \text{const} = 1$.

$$T(n) = 2 \sum_{k=1}^{n-1} T(k) + O(n)$$

$T(n) \geq U(n)$

let $U(n) = 2[U(n-1)] + 1$

$U(n) \geq 2^n - 1$

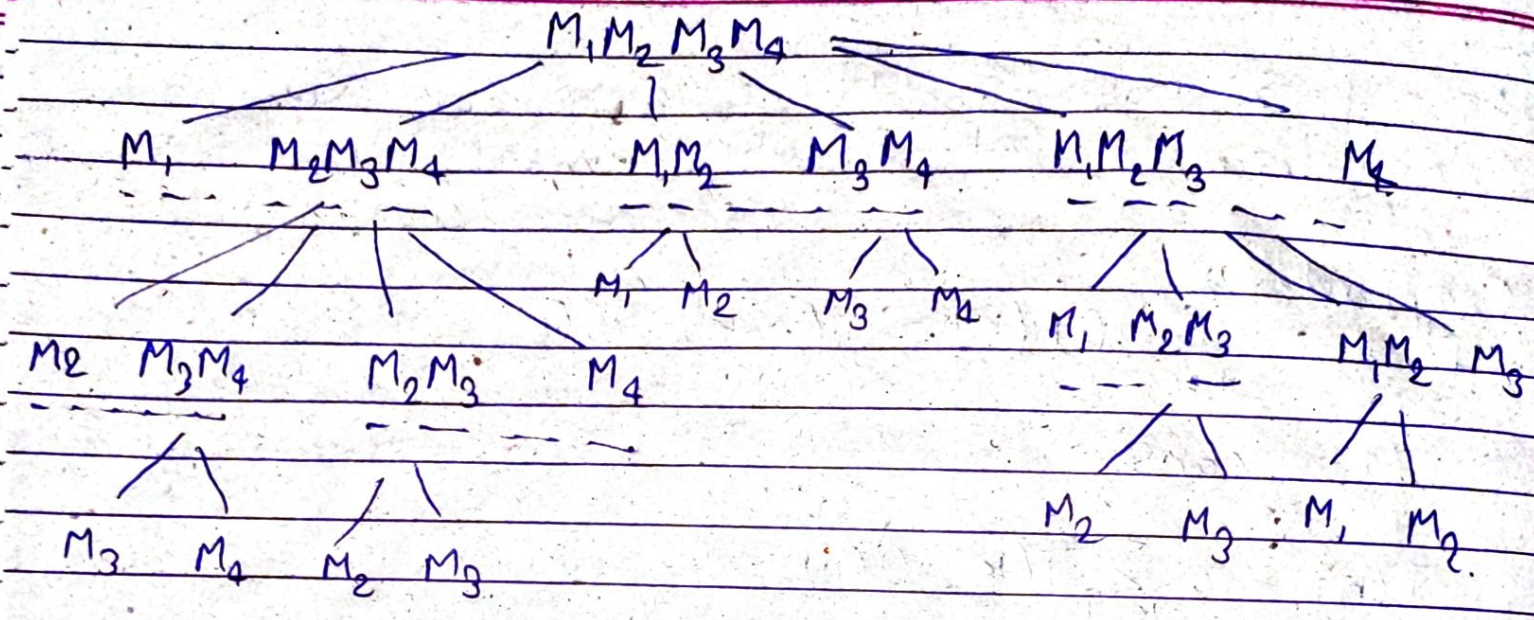
$U(n) \geq 2^n$

$T(n) \geq 2^n$

Thus complexity of recursive approach is exponential

* Using ~~Top-Down DP (Memoization)~~
Bottom-Up (Tabulation)

lets Suppose we have four matrix M_1, M_2, M_3, M_4
Based on the recursive approach described above, we can construct a recursion tree.
However we can observe some problems are computed multiple time.




```
Procedure Matchain(P)
    for i=1 to n do M[i,i]=0 end for
    for l = 2 to n do
        for i=1 to n-l+1 do
            j = i+l-1 ; M[i,j] = -∞
            for k=i to j-1 do
                mult = M[i,k] + M[k+1,j] +
                P[i-1] * P[k] * P[j];

                if mult < M[i,j] then
                    M[i,j] = mult ; S[i,j] =
                end if
            end for
        end for
    end for
```

3 Nested Loops $O(n^3)$

P	5	4	6	2	7

A_1 A_2 A_3 A_4

5×4 4×6 6×2 2×7

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

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

$m[1,1] = A_1 = 0.$

$m[1,2] = A_1 \cdot A_2$

$5 \times 4 \quad 4 \times 6 = 120.$

$m[2,3] = 4 \times 6 \times 2 = 48$

$m[3,4] = 6 \times 2 \times 7 = 84$

$m[1,3] = A_1 (A_2 \cdot A_3)$

$5 \times 4 \times 2$

$= 40$

$= m[1,1] + m[2,3] + 5 \times 4 \times 2$

$= 0 + 48 + 40$

$= 88$

$(A_1 \cdot A_2) A_3$

$5 \times 6 \times 2$

$= 60$

$= m[1,2] + 5 \times 6 \times 2$

$= 120 + 60$

$= 180$

$m[2,4] = A_2 (A_3 A_4)$

$= 84 + 7 \times 4 \times 6$

$= 252$

$(A_2 A_3) A_4$

$= 48 + 4 \times 2 \times 7$

$= 104$

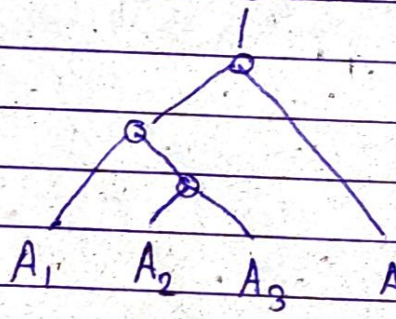
$$m[i,j] = m[i,k] + m[k+1, j] + d_{i-1} \times k_t \times d_j \}$$

$$m[1,4] = \min\{ m[1,1] + m[2,4] + 5\times4\times7,$$
$$m[1,2] + m[3,4] + 5\times6\times7,$$
$$m[1,3]\ m[4,4] + 5\times2\times7 \}$$

$$= \min\{ 244.$$
$$= 188$$

$(A_1\ A_2 \cdot A_3)(A_4)$

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



2) P = | 4 | 2 | 5 | 1 | 3 |

A_1 : 4×2 , A_2 : 2×5 , A_3 : 5×1 , A_4 : 1×3

m	1	2	3	4
1	0	40	18	30
2		0	10	16
3			0	15
4				0

S.	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

$m[1,3] = \min \{ m[1,1] + m[2,3] + 4 \times 2 \times 1,$
$m[1,2] + m[3,3] + 4 \times 5 \times 1 \}$

$= \min \{ 10 + 8, 40 + 20$
$= 16$

$m[2,4] = \min \{ m[3,4] + 2 \times 5 \times 3$
$m[2,3] + 2 \times 1 \times 3 \}$

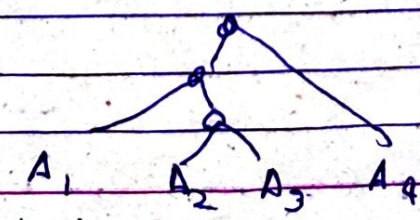
$= \min \{ 15 + 30,$
$10 + 6$
$= 16$

$m[1,4] = \min \{ m[1,1] + m[2,4] + 4 \times 2 \times 3,$
$m[1,2] + m[3,4] + 4 \times 5 \times 3,$
$m[1,3] + m[4,4] + 4 \times 1 \times 3 \}$

$= \min \{ 16 + 24,$
$15 + 60,$
$18 + 12$
$= \boxed{30}$

Tracking (Track Point)

$((A_1)(A_2 \cdot A_3)) \cdot (A_4)$



* Time Complexity.

We are filling elements in half of the table $\Rightarrow \frac{n(n+1)}{2}$.

and we are finding minimum for each element.

$$\Rightarrow \frac{n(n+1)}{2} \times n$$

$$\doteq O(n^3)$$

* Space Complexity.

$O(n^2)$ $\longrightarrow$ Grid Generated.

```
print_parenth (S, i, j)
  if i = j print "Ai"
  else
      print "("
      print_parenth (S, i, S[i, j])
      print_parenth (S, S[i, j]+1, j)
      print ")"
  end if
```

* Rod Cutting Problem:

- The rod cutting problem is a classic dynamic programming (DP) problem where the goal is to determine the best way to cut a rod into smaller lengths to maximize revenue

- Given a rod of length n and a price array price[], where price[i] represents the revenue obtained by selling a rod of length i+1, determine the maximum revenue that can be obtained by cutting the rod into pieces of various lengths.

* Recursive Approach:

The recursive approach involves solving the problem by considering all possible ways to cut the rod into two pieces at every length j where (1<=j<i).

A rod of length n can be cut in 2^{n-1} different ways since we can choose cutting or not cutting at all distances i ($1 \le i \le n-1$) from left end.

In case of recursive approach we can calculate the maximum value r_n in terms of optimal values from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots r_{n-1} + r_1)$$

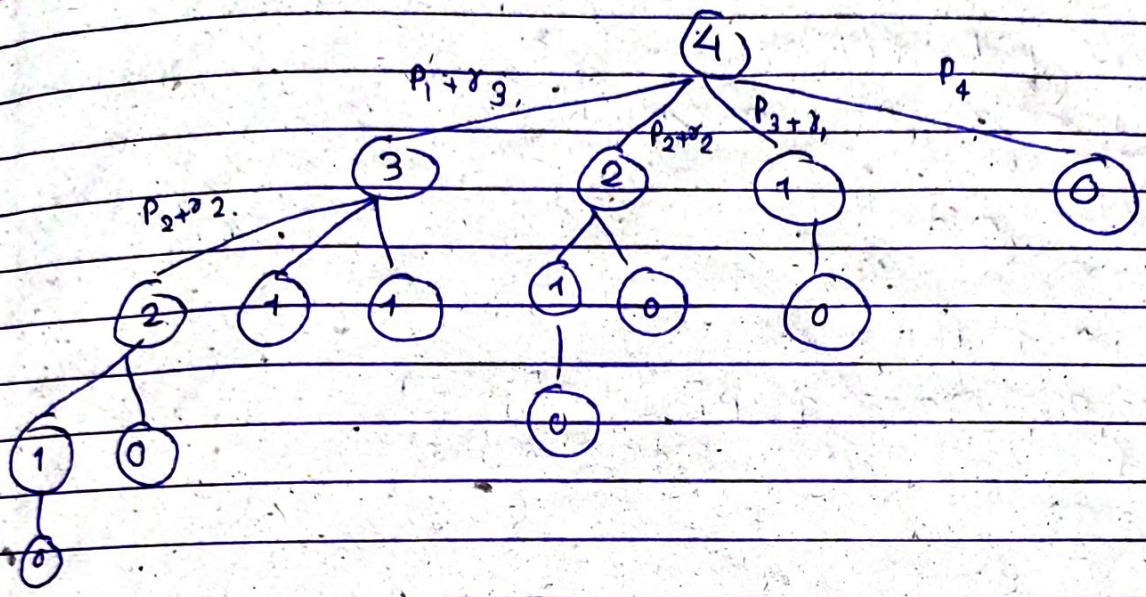
- The first argument, p_n corresponds to making no cuts at all and selling the rod of length n as is.
- The other n-1 arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and n-i for each i = 1, 2 .. n-1 and the optimally cutting up those pieces further obtaining revenues r_i and r_{n-i}

Thus

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

```
Cut_Rod (p, n)
if n == 0 then
    return 0;
q = -∞
for i = 1 to n do
    q = max (q, p[i] + Cut_Rod (p, n-i))
endfor
return q
```

The procedure Cut_Rod takes as input an array p[1..n] of prices and an integer n, and it return the maximum revenue possible for a rod of length n.



$T(n)$ = Total calls made to cut rod function with rod length 'n'.

= No of nodes in a subtree whose root is labeled n in recursion tree.

$$T(n) = \begin{cases} 1 + \sum_{i=0}^{n-1} T(i) & \\ 1 \quad \text{if } n = 0 & \end{cases}$$

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

* Dynamic Programming using bottom up approach.
- In this approach we solve each subproblem only once and store its solution in a db table.
- If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it.

```
Bottom_Up_Cut_Rod(P,n)
let r[0..n] be a new array.
r[0] = 0;
for j = 1 to n do
    q = -∞
    for i = 1 to j do
        q = max(q, P[i] + r[j-i])
    end for
    s[j] = i // store size of ith piece
    r[j] = q
end for
return r[n]
```

```
Print_Cut_Rod_Solution(P,n)
while n>0 do
    print s[n]; // Print Size of pieces
    n = n - s[n];
end
```

```
Bottom Up-Cut-Rod(P,n)
let r[0,n] and S[0..n] be new arrays
r[0] = 0.
for j = 1 to n
    q = -∞
    for j=1 to n
        q = -∞
        for i=1 to j
            if q < P[i] + r[j-i]
                q = P[i] + r[j-i]
                s[j] = i
            end if.
        end for
    r[j] = q.
```

return r and s

length i	1	2	3	4	5	6	7	8	9	10
price pi	1	5	8	9	10	17	17	20	24	30

length i	0	1	2	3	4	5	6	7	8	9	10
price pi	0	1	5	8	9	10	17	17	20	24	30
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

* Amortized Analysis:
- Amortized Analysis is a technique used in algorithm analysis to determine the average time per operation over a sequence of operations, even though a single operation might be expensive.
- It helps us prove that certain operations run efficiently over time, even if some individual operations seem costly.

- Amortize: Gradually write off the initial cost over a period."

* Binary Counting Problem:
By applying traditional approach, complexity is O(n log n)
- If trailing zero is obtained from right to left, flip it to 1 and stop.
- If trailing one is obtained flip it and next flip the first one that is obtained moving from right to left.
- If two consecutive trailing zero is obtained flip both to one and the flip the next zero to one.

flip all consecutive 1s (trailing) to 0 and flip 0 to 1 and Stop.

```
Procedure Increment
    i = 0
    while A[i] = P do
        A[i] = 0;
        i = i + 1;
    end while
    A[i] = 1;
```

* Acounting Method : (Banker's)
  - This analysis is based on amortized cost.
  - If the amortized cost exceeds the actual cost, the excess remains with data structure as a credit.
  - This credit can be used to pay for future expensive operations.

Assume

0 → 1 2Rs
1 → 0 0Rs

A[3]	A[2]	A[1]	A[0]	
0	0	0	0	→ 2Rs
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	

At each stage we are utilizing 2rs and this goes till n so time complexity = O(2n) = O(n)

- We use a charge of 2 ~~dollars~~ Rs to set a bit to 1.
- When a bit is set, we use 1 Rs (out of 2) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- At any point in time every 1 in the counter has a Rs of credit on it, thus we can charge nothing to reset a bit to 0.

- If the amortized cost is small enough so that the actual cost cannot be covered then, it is paid by the credit.
- Each increment has amortized cost of 2 Rs so that there are atmost 2n bit changes.

* Potential Method:

The potential method is another way of performing amortized analysis, similar to the accounting method, but instead of assigning credits, it uses a mathematical function (potential function) to measure the system's stored energy. (potential of data structure)

The potential method works as follows. We perform n operations, starting with an initial data structure D_0.

For each $i = 1, 2 \ldots n$

C_i is the actual cost of i^{th} operation.

D_i is the data structure after i^{th} operation.

- A potential function ϕ maps each data structure D_i to a real number $\phi(D_i)$.

$\phi(D_i)$ = Potential of D_i.

- The amortized cost a_i of the i^{th} operation with respect to potential function ϕ is defined by

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

a_i = actual cost + change in potential due to operation.

Total amortized cost of n operations is.

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

If we choose a potential function ϕ such that $\phi(D_0) = 0$ and $\phi(D_n) \geq 0$ then.

$$\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} c_i$$

amortized cost is upperbound for the actual cost.

$$\phi(D_i) = b_i$$

b_i indicates how many total 1's are present in bit stream.

$$\phi(D_i) - \phi\,\phi(D_i - 1) = b_i - b_{i-1}$$
$$= (b_{i-1} - t_{i-1} + 1) - b_{i-1}$$
$$= 1 - t_{i-1}$$

where t = trailing ones

$$c_i = t_{i-1} + 1.$$

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$$
$$= t_{i-1} + 1 + b_i - b_{i-1}$$
$$= t_{i-1} + 1 + 1 - t_{i-1}$$
$$= 2$$

Thus amortized cost of push operation is 2

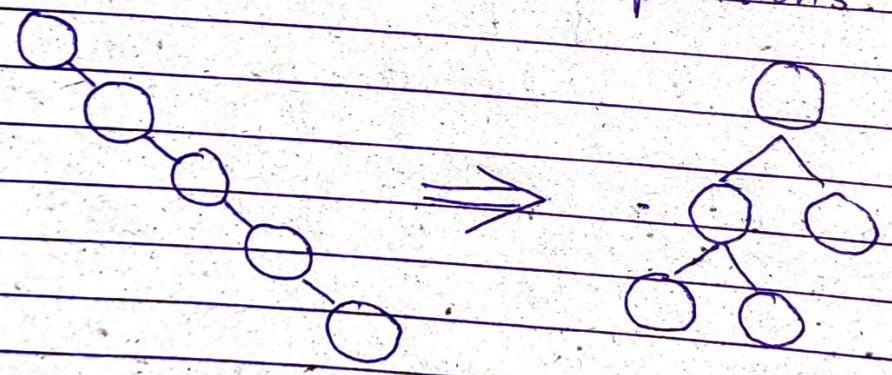
for n number of operator.

$$\sum_{i=1}^{n} a_i = 2n$$

$\Rightarrow$ 2n is upper bound for actual cost.

★ Red Black Tree (RB Tree)

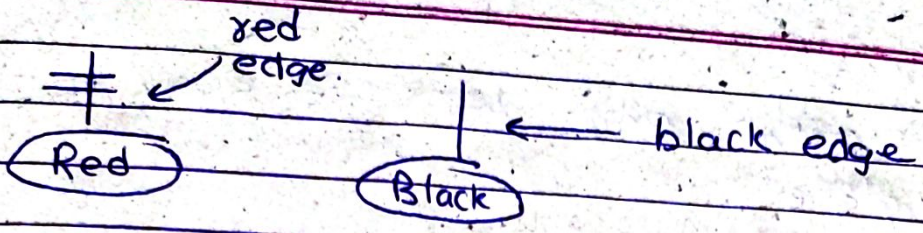
- A Red-Black Tree is a self balancing binary search tree that maintains balance using color properties and rotations.
- It ensures O(log n) time complexity for insertion, deletion and search operations.



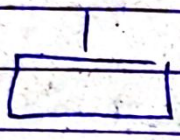
Rotation Operation

- Single Rotation
- Double Rotation

Left	Color	Key	Ptr to parent	Right

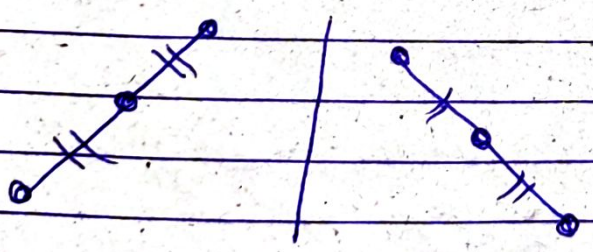


- Leaf Nodes are empty pointers.



★ Properties.

- Every node is either red or black.
- The root node is always black
- No two consecutive red nodes.
- Every path from a node to its descendant NULL nodes must have the same number of black nodes. (Same black height). bh(u)
- Newly inserted nodes are always red.



Consecutive Red edges (Not Allowed)

★ Insertion in RB Tree.

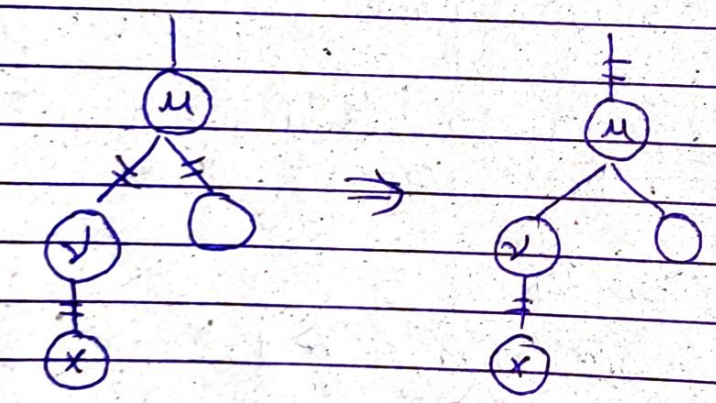
- When inserting a new node in RB-tree, it is initially colored red.
- However, this may violate the R-B tree property, So we must apply recoloring and rotations to restore balance.

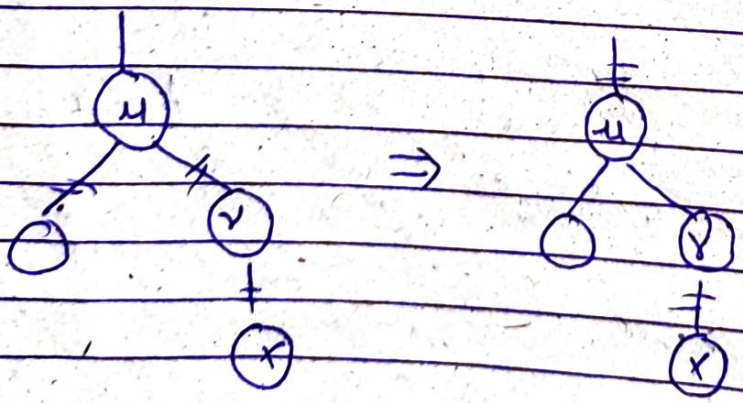
to which new node.
x is node ~~that~~ is inserted

* Case 1: Incomming edge to x is black
  → No Violate
  done.

* Case 2: If incomming edge to x is red:
  Set u = Parent of x.

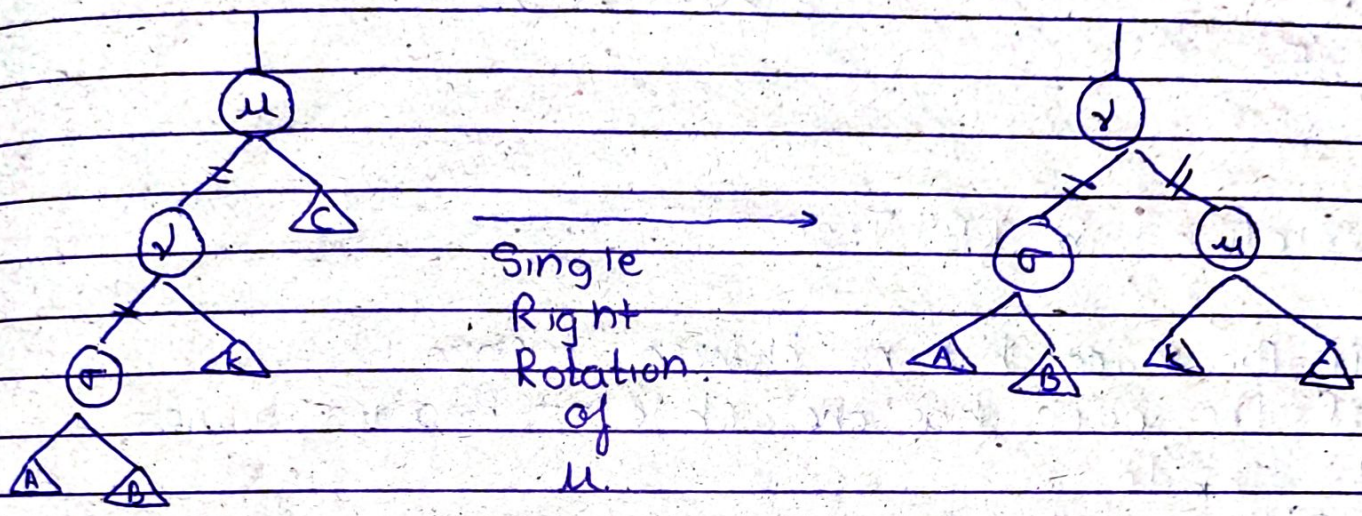
  Case 2.1: Both outgoing edge of u are red.
  ↳ Promote u if $u.p \neq Nil$
  [Increase black height]

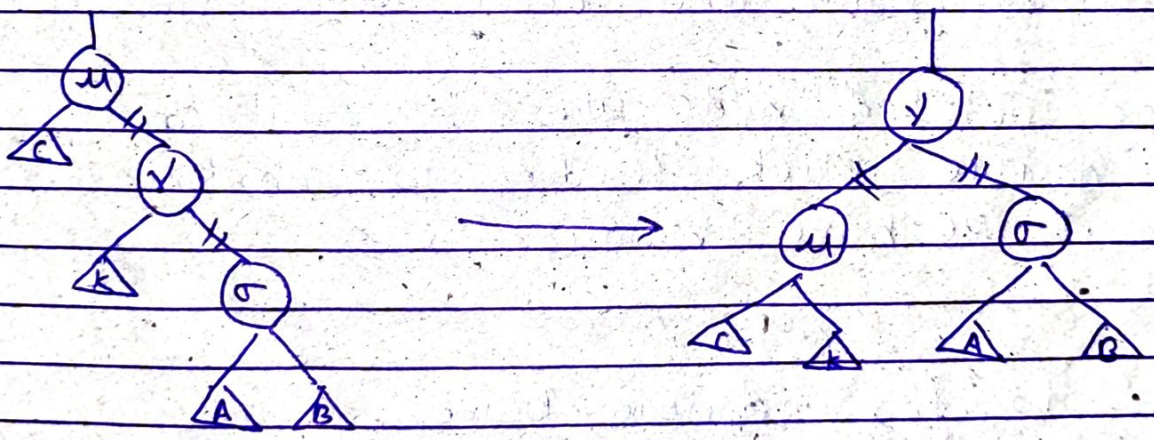




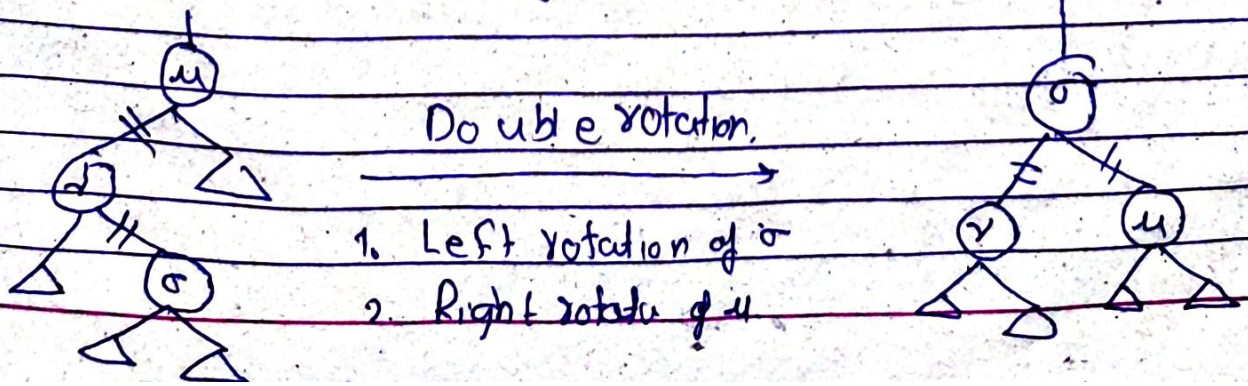
Making this change can lead to violation by $u.p$ so we set $x = u.p$ and recurse.

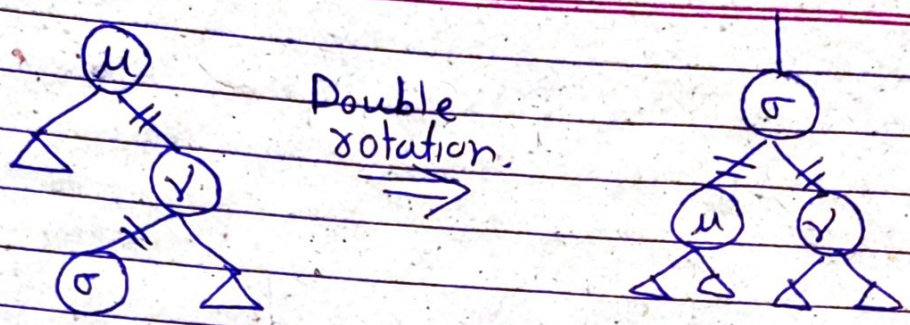
Case 2.2: Only one child of u is red.





Case 2.3: v is left child of u and right outgoing edge of v is red.





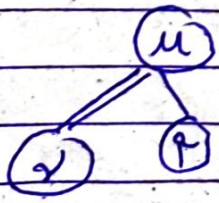
* Deletion in RB Tree.

1. If the node has no children remove it
2. If the node has one child → Replace it with its child.
3. If the node has two children
   - find inorder successor
   - Swap value and delete it.

* If the deleted node was black, it can cause double black, leading to imbalance. To fix it we have various cases.

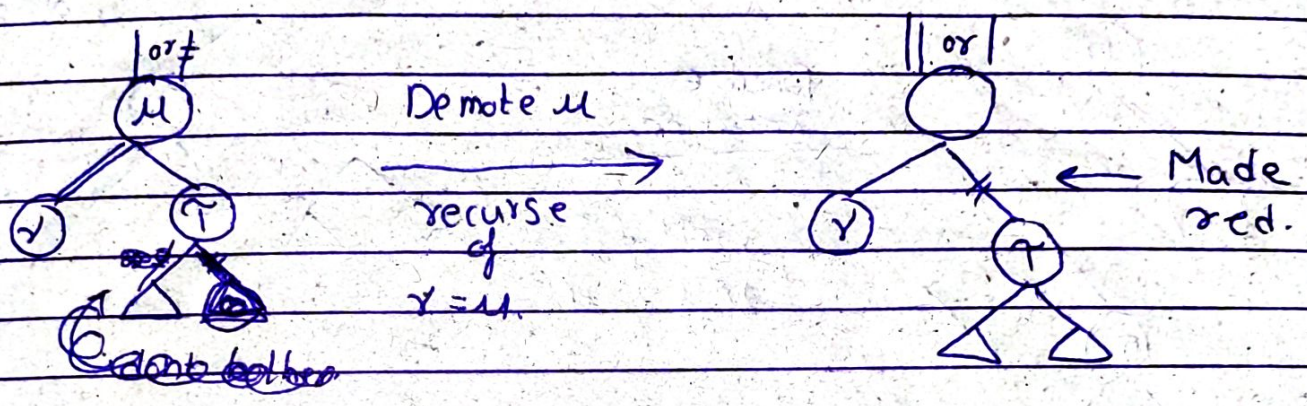
* Cases to fix double black.

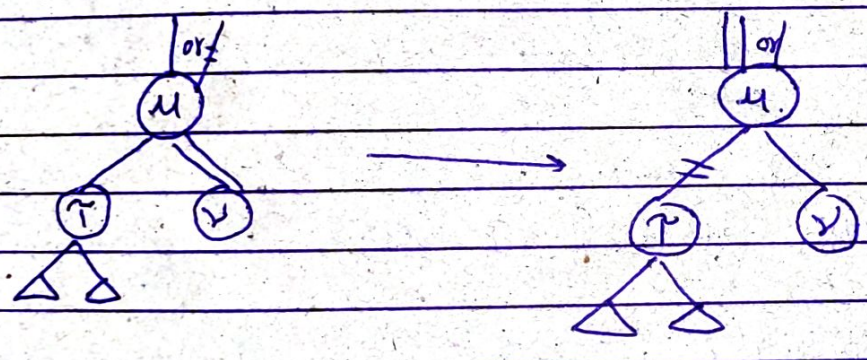
Case 1: Incoming edge of v is double black
- u = v.p
- τ is sibling of v.



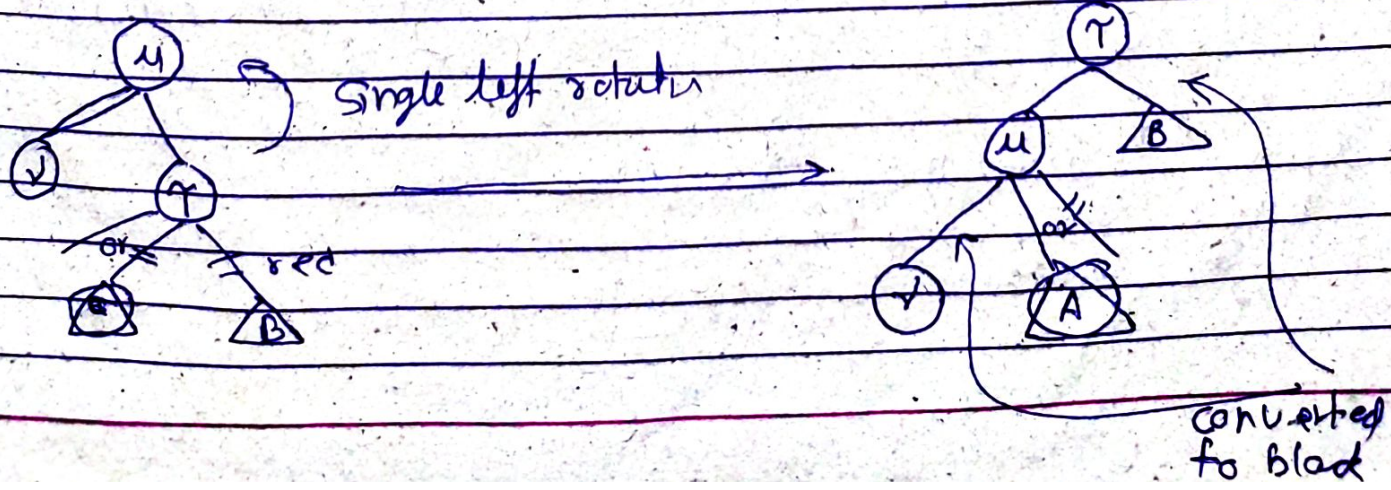
Case 1.1 : τ is not a leaf

Case 1.1.1 : Both outgoing edges of τ are black.

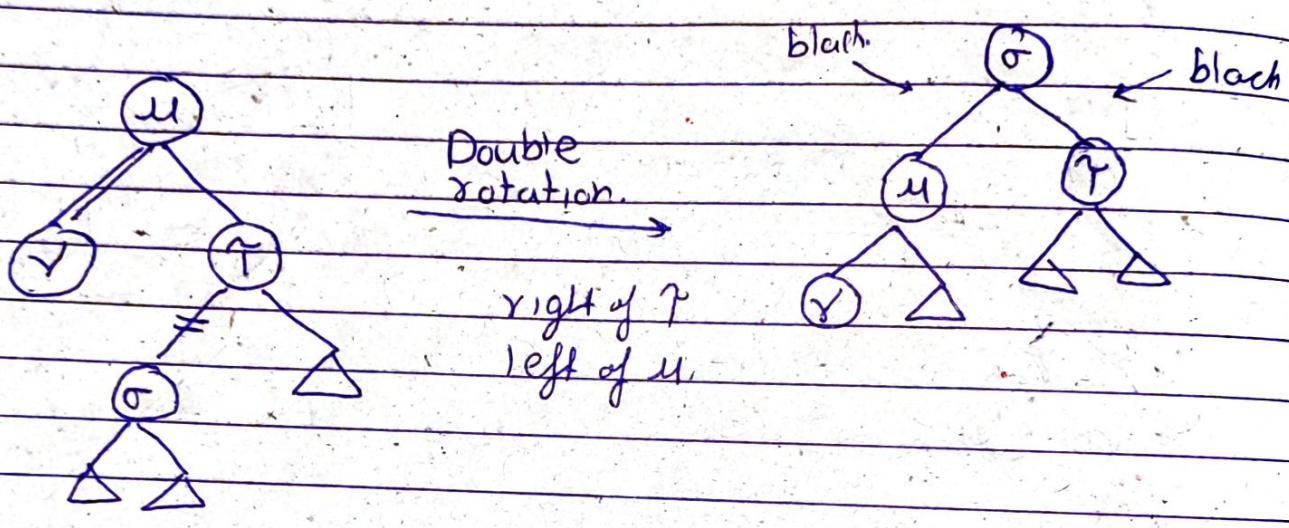




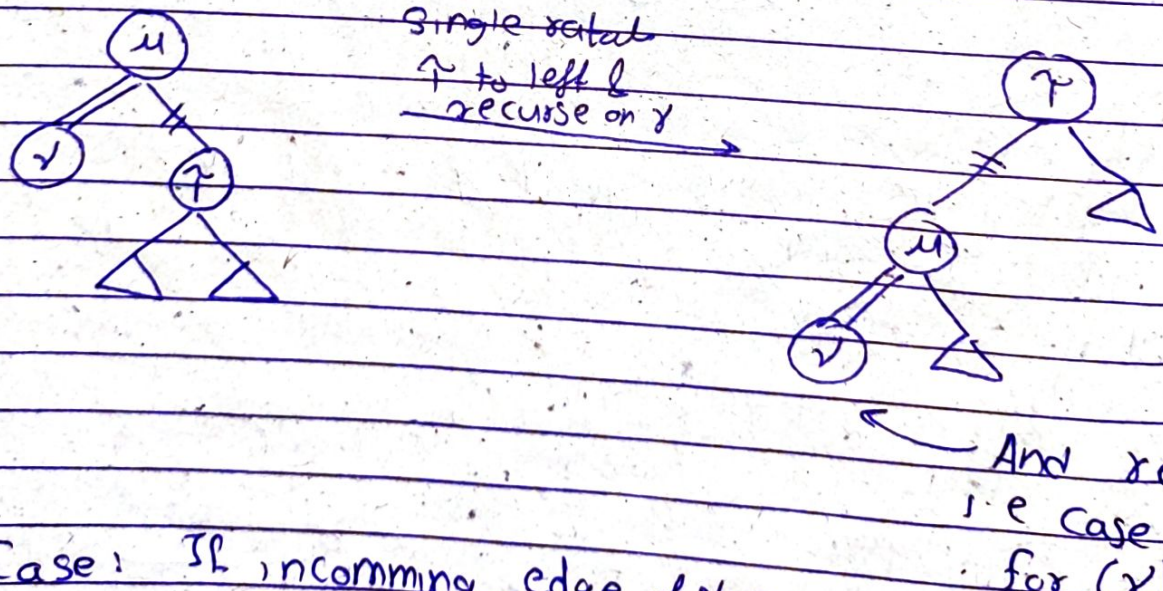
Case 1.1.2 : v is the left Child of u and ~~right~~ outgoing of τ is red ~~the other is black~~



Case 1.13 : v is the left child of u and left outgoing edge of τ is red the other one is black.



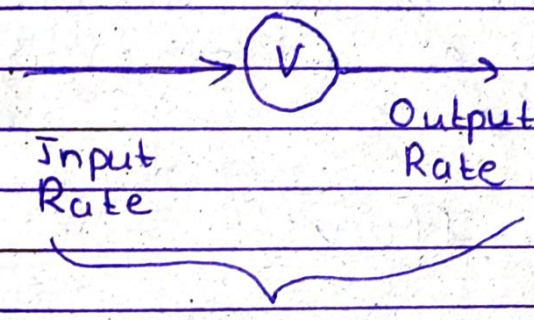
* Case 1.2 : Edge from u to τ is red.



* Case : If incomming edge of v is black.
⇒ We have done STOP

* Flow Networks:

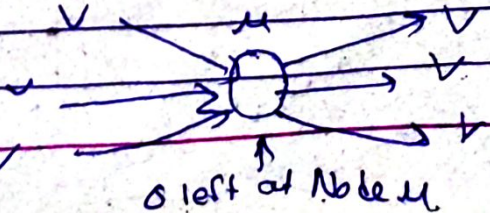
- A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non negative capacity $c(u, v) \geq 0$.
- Edges are present only in one direction.
- If $(u, v) \notin E$ then $c(u, v) = 0$.
- Self loops are not allowed.
- We define two vertices
  (s) 1. Source : produces material at same steady rate
  (t) 2. Sink : Consumes the material at same rate.



- Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source and t be the sink of the network.
- A flow in G is a real valued function
  $f : V \times V \longrightarrow \mathbb{R}$
  that satisfies the following.
  i) Capacity constraint : For all $u, v \in V$, we require
  $0 \leq f(u, v) \leq c(u, v)$.

  ii) Flow conservation : For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$



* Maximum Flow Problem:
We wish to compute the greatest rate at which we can shift material from source to sink without violating any capacity constraints.

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

* Residual Capacity:
It is the additional amount of flow that can still be sent through an edge without exceeding its capacity.

$$r(u,v) = c(u,v) - f(u,v)$$

* Max Flow Min Cut Theorem:
In a flow network, the maximum amount of flow that can be sent from source (s) to the sink (t) is equal to the total capacity of the minimum cut that separates s and t.

- A cut in a flow network is a partition of vertices into two disjoint sets.
  - One set contains the source
  - The other contains the sink t.

- The capacity of a cut is the sum of the capacities of edges crossing from source set to the sink.
- The minimum cut is the cut with the smallest capacity that separates s and t.

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

If f is a flow in flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S,T)$ for some cut (S,T) of G.

* Ford - Fulkerson Method:

- The ford-fulkerson method finds the maximum flow in a flow network using augmenting paths and residual graphs.
- It iteratively increases the value of flow starting from $f(u,v) = 0$ for all $u, v \in V$
- At each iteration we increase the flow value in G by finding augmenting path in associated residual network G_f.
- We repeatedly augment the flow until the residual network has no more augmenting path

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

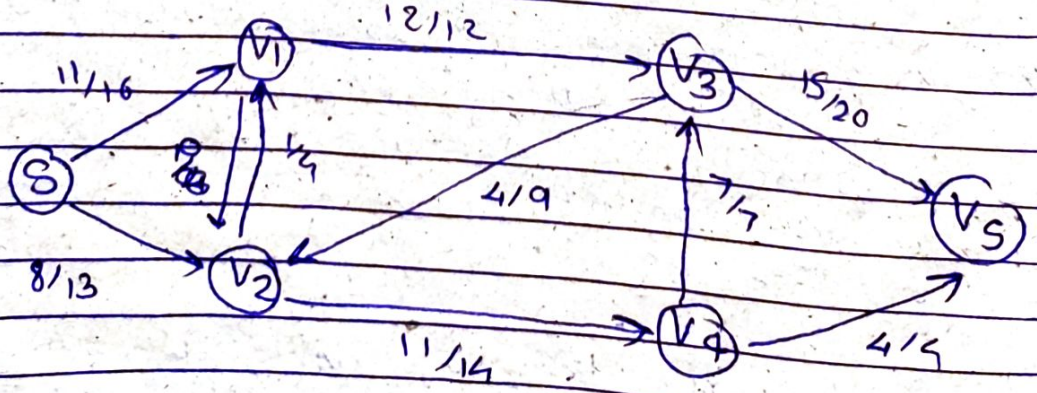
c_f = Residual Capacity.
c = Capacity of edge / link.

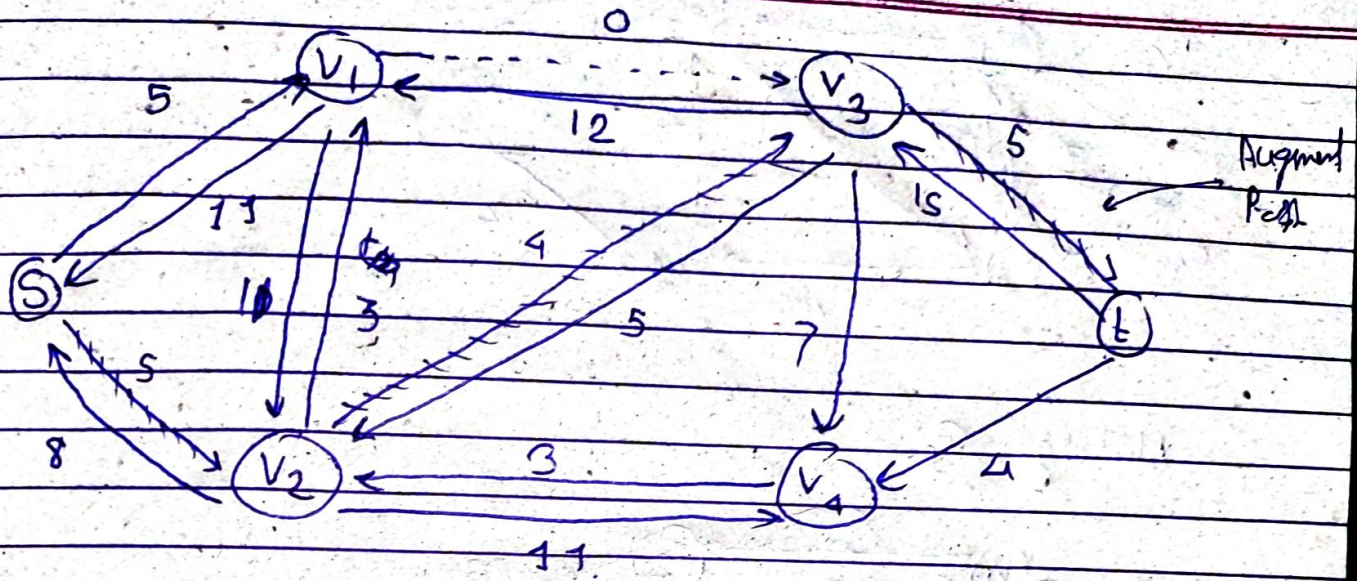
Ford-Fulkerson-Method (G, s, t)
initialize flow f to 0
while there exist an augmenting path p in G_f
augment flow f along p
return f.

Give a flow network $G = (V, E)$ and a flow f, the residual network of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{ (u,v) \in V \times V : c_f(u,v) > 0 \}$$

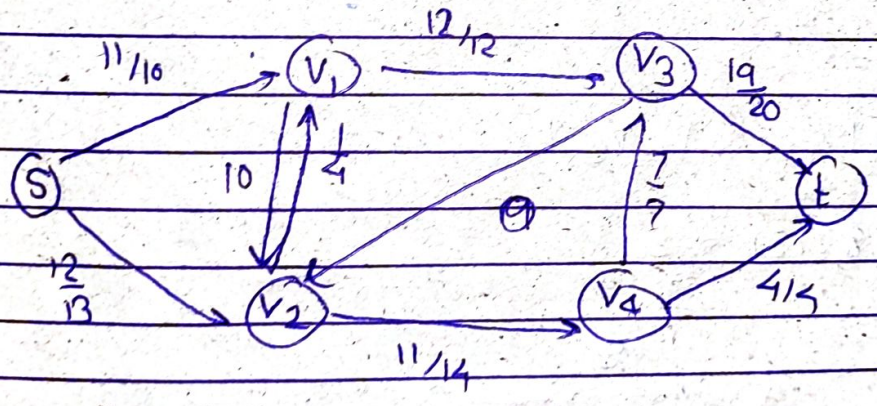
Ex.

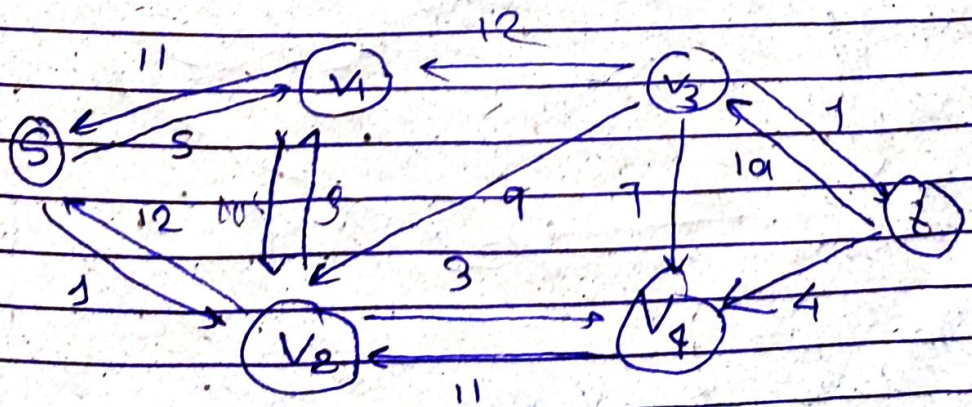




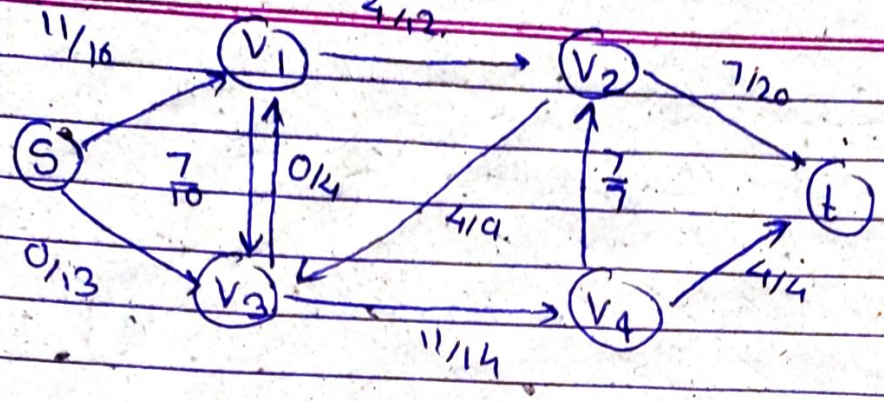
Residual Network G_f with augmenting path p shaded.

min(5, 4, 5) = 4.

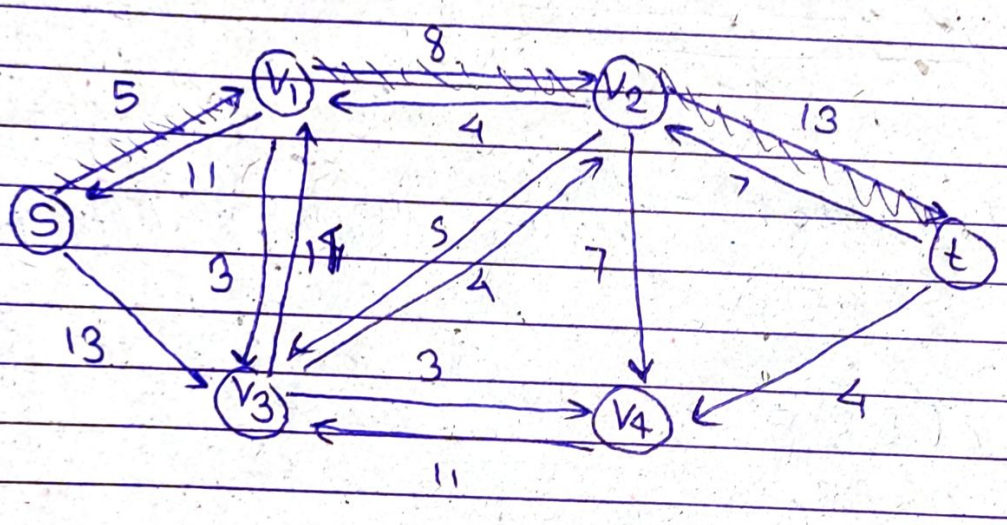




Answer Q3



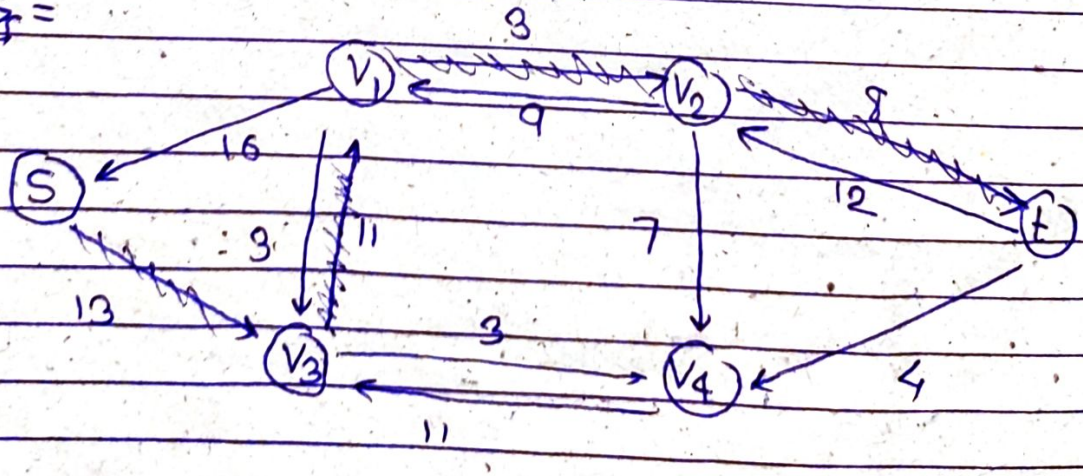
Making G_f



Taking Path $S V_1 V_2 t$

Min [5, 8, 13] = 5.

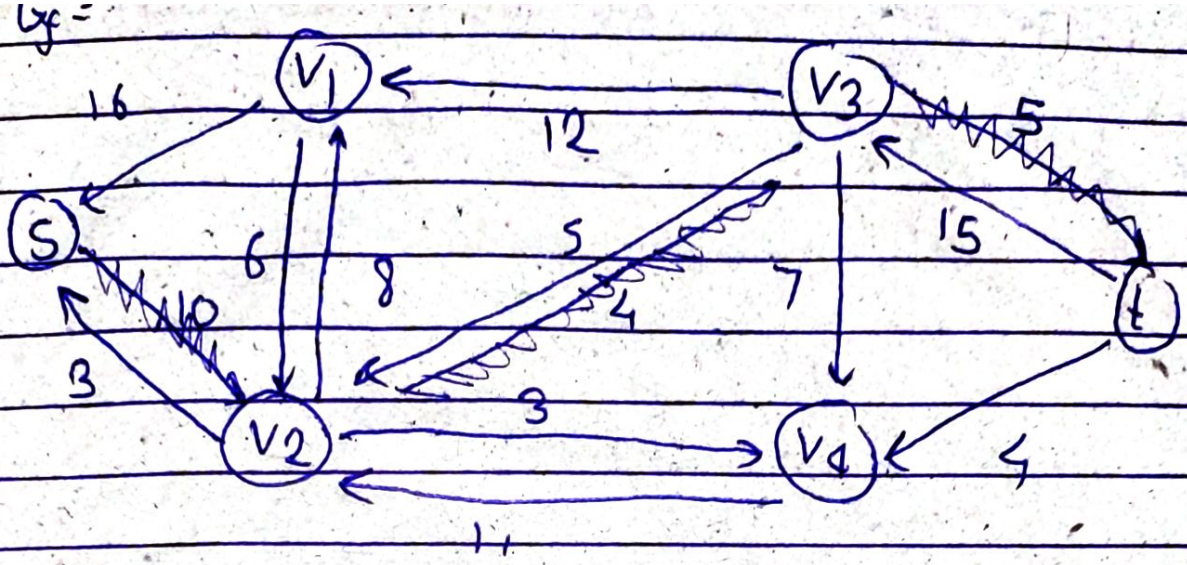
Updated G_f =



Taking Path $S V_3 V_1 V_2 t$

min [13, 11, 3, 8] = 3.

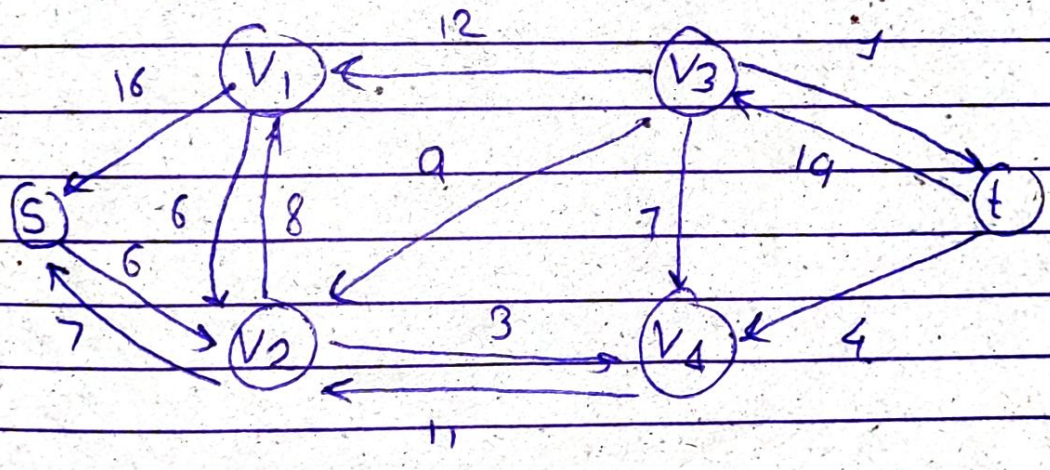
Updated G_f =



Taking Path $S V_2 V_3 t$

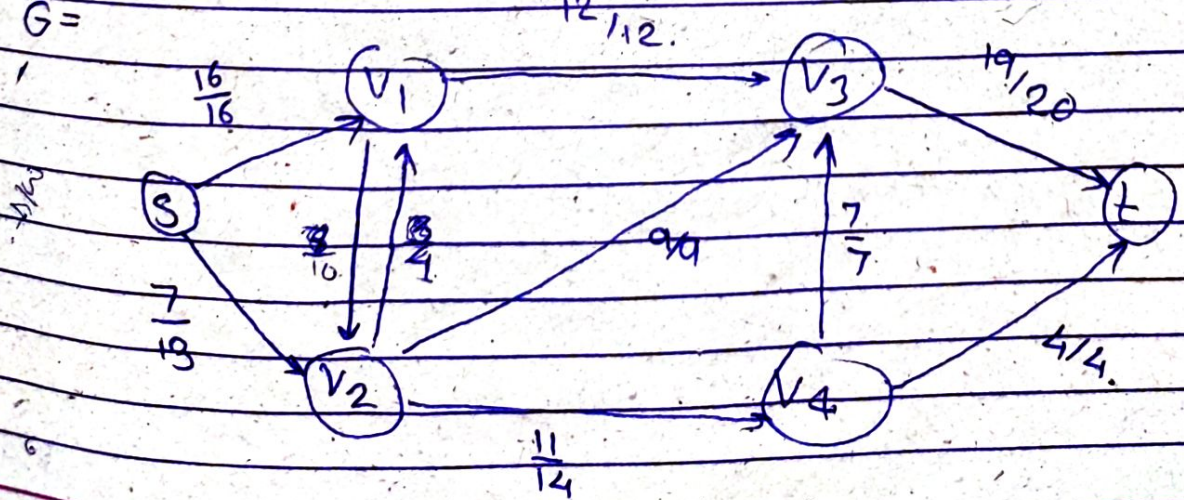
$\min(10, 4, 5) = 4$

Updated G_f



No more augmeting Path available.

G =



Ans = 23