Dynamic Programming

Rajkishor Ranjan

rraju121295@gmail.com

July 18, 2025

Introduction

- Dynamic Programming, Importance and uses in many Problems.
- Fibonacci Number,Longest Palindrome Subsequence , Matrix Chain Multiplication, Boolean Parenthesization Problem etc
- Fibonacci Number : A sequence of numbers in which each number is the sum of the two preceding numbers. Ex. 0, 1, 1, 2, 3, 5, 8, 13...etc.
- **Problem:** Write a code to find the n^{th} Fibonacci Number.

Fibonacci Number:

Sequence : 0,1,1,2,3,5,8,13,21,34,55.......

Recursive Relation (nth term of the sequence)

Except the 1st and 2nd terms, we see that every term is the sum of it's 1st two preceding terms. i.e. If we think about the nth term then

nth term = (n-1)th term + (n-2)th term $\forall$ n $\geq$ 2

Base Case : nth term = 0 if n=0 and nth term =1 if n=1

Pseudo code :

```
int Fib(int n)
    if(n==0)
    return 0;
    else if(n==1)
    return 1;
    else
    return Fib(n-1)+Fib(n-2);
```

Recursive Code

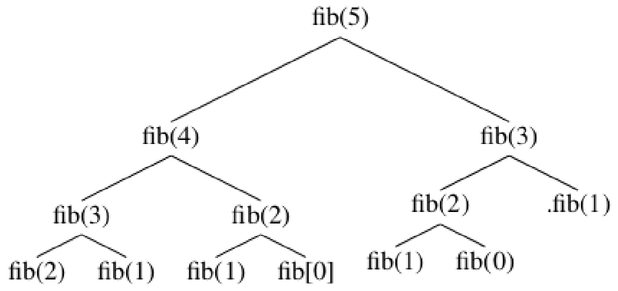
```cpp
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

//Function to find Fibonacci Number
int fib(int n)
{
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
    {
        return fib(n-1)+fib(n-2);
    }
}

int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        if(n<0)
            cout<<"Give input greater than 0";
        else
            cout<<n<<"th Fibonacci Number is :"<<fib(n)<<"\n\n";
    }
```

Output :

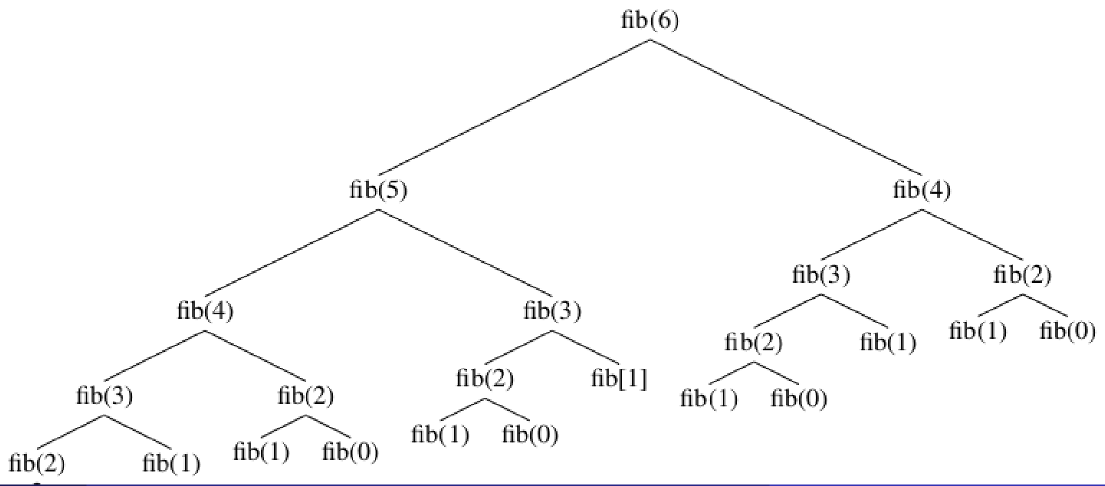
```
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC: ~/Documents
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~$ /opt/sublime/Packages
bash: /opt/sublime/Packages: No such file or directory
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~$ cd Documents/java_code/Unacade
my_Code/
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ ls
Fibonacci_2       Fibonacci_3       Fibonacci_Number       Input.txt
Fibonacci_2.cpp   Fibonacci_3.cpp   Fibonacci_Number.cpp   Picture
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ g++ Fibonacci_Number.cpp -o Fibonacci_Number
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ ./Fibonacci_Number
3
9 14 15
9th Fibonacci Number is :34

14th Fibonacci Number is :377

15th Fibonacci Number is :610

rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$
```

Recursion Tree

for n = 5



for n = 6



Time Complexity

- Some functions are called many times.

Recurrence relation for time complexity

T(n) = T(n-1) +T(n-2) +O(1)
i.e. T(n) = O(2^n)

Problem : How can we avoid these multiple overlapping subproblems?
Solution : If we store the value of subproblems (like fib(5),fib(4), fib(3), fib(2) ,... etc.), then instead of computing it again and again, we can use the old stored value. isn't it?

Top down(Memoization)

Kadane_Alç Square_SubMatrix.cpp × Fibonacci_Number.cpp ×

```cpp
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
#define NIL -1
using namespace std;

int fib_num[MAX];
int fib(int n)
{
    if(fib_num[n]==-1)
    {
        if(n==0)
          fib_num[n]=0;
       else if(n==1)
        fib_num[n]=1;
       else
        fib_num[n]=fib(n-1)+fib(n-2);
        return fib_num[n];
    }
    else
     return fib_num[n];

}
```

Kadane_Alç Square_SubMatrix.cpp × Fibonacci_Number.cpp × Fibonacci_2.cpp × Fibonacci_3.cpp ×

```cpp
}
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        for(int j=0;j<=n;j++)
        {
            fib_num[j]=-1;
        }
        cout<<n<<"th fibonacci number is equal to :"<<fib(n)<<"\n

    }
}
```

Bottom Up (Tebulation)

```cpp
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
int fib(int n)
{
    int fib_num[n+1];
    for(int i=0;i<=n;i++)
    {
      if(i<=1)
        fib_num[i]=i;
      else
        fib_num[i]=fib_num[i-1]+fib_num[i-2];
    }
    return fib_num[n];
}
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        cout<<n<<"th fibonacci number is equal to :"<<fib(n)<<"\n\n";
    }
}
```

Dynamic Programming(DP)

- Solve the main problem by combining the solutions to subproblems.
- It solves each subproblem just once and save the result of each subproblem.
- It applies when the subproblems overlap i.e. when subproblems share subsubproblems.

Sequence of steps when we developing a Dynamic Programming

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution(Top Down or Bottom Up)

Dynamic Programming (DP)

There are two main properties of the problems where we can use the concept of Dynamic Programming.

1. Overlapping Subproblems

- Solution of subproblems needed again and again.
- In DP, we store the result of each computed subproblem in a table(mainly in array or matrix)
- DP is not useful when there is no common or overlapping subproblems.
  Ex. Binary Search : Here there is no overlapping subproblem.

2. Optimal Substructure

We need optimal solution of the given problem by using optimal solutions of its subproblems.

Optimal Substructure

Problem : There are many cities between the two city A and B. We need to find the shortest distance between these two cities A and B.
Approach : If a city X lies in the shortest path from city A to city B then the shortest path from A to B is combination of shortest path from A to X and shortest path from X to B.

How can we store the result of Subproblems in DP?

There are following two different ways to store the values so that the values of a problem can be reused.

1. Top Down (Memoization)

- We store the result of subproblems in the memory whenever we solve the problem for the 1st time and next time we simply do a lookup,
- It is similar to the recursive version with a small modification that it looks into a lookup table(generally array or matrix) before computing solutions.
- If the precomputed value is present in the lookup table then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later (in case of overlapping subproblem).

2. Bottom Up (Tabulation)

- According to the name it start to solve the problem from the bottom(base state) and cumulating answers to the top (most desired state).
- It precomputed the solutions in a linear fashion and store it in a table.

Difference between Top Down and Bottom Up

- In **Bottom Up**, we start from smallest instance size of the problem, and ITERATIVELY solve bigger problems using solutions of the smaller problems(i.e. reading from the table).
- In **Top Down**, we start from the original problem, and solve it by breaking it down into subproblem(using RECURSION). When we have to solve subproblem, we first check in a look-up table to see if we already solved it. If we did, we just read it up and return value without solving it again.Otherwise, we solve it recursively, and save result into table for further use.

Longest Palindrome Subsequence

Problem :

Given a sequence of character, find the length of the longest palindromic subsequence in it.
Ex. if the given sequence is “BBABCBCAB”, then the output should be 7 as “BABCBAB” is the longest palindromic subsequence in it.

For given sequence “BBABCBCAB”, subsquence ”BBBBB” and ”BBCBB” are also palindromic subsequnce but these are not largest.

Approach : The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence.

Optimal Substructure Property :

Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].

$$X = A_1A_2A_3....A_{n-1}A_n$$

Recursive Relation :

$$L[i,j] = \begin{cases} 2 + L[i+1, j-1], & \text{if } X[i] = X[j] \\ max(L[i, j-1], m[i+1, j]), & \text{if } X[i] \neq X[j] \end{cases}$$

Base Cases :

$$L[i,j] = \begin{cases} 1, & \text{if } i = j \\ 2, & \text{if } i = j-1, X[i] = X[j] \end{cases}$$

Start the code with $i = 0$ and $j = n - 1$;

Recursive Code

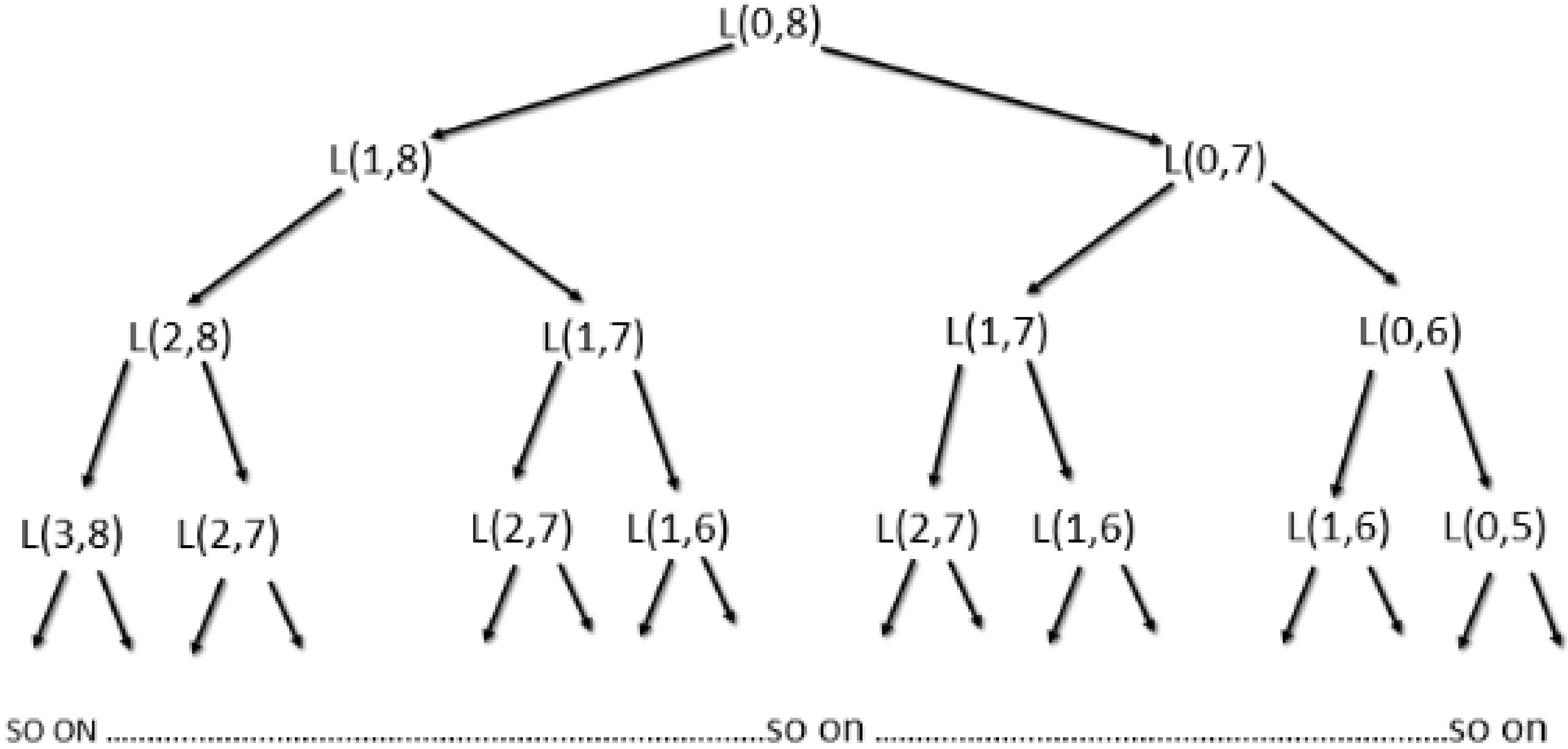
```cpp
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int SubSeq(char seq[], int i, int j)
{
    int length_SubSeq =0;
   if(i==j)
    return 1;
   else if(i==(j-1) && seq[i]==seq[j])
   {
        return 2;
   }
   else if(seq[i]==seq[j])
    length_SubSeq = SubSeq(seq,i+1,j-1)+2;
   else
    length_SubSeq = max(SubSeq(seq,i+1,j), SubSeq(seq, i, j-1));
   return length_SubSeq;
}
```

```cpp
}

int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {

        int n;
        cin>>n;
        char seq[n];
        for(int j=0;j<n;j++)
        {
            cin>>seq[j];
        }
         cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,0,n-1)<<"\n"
    }
}
```

Output

```
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ g++ Max_Palinrome_Subseq.cpp -o Max_Palinrome_Subseq
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ ./Max_Palinrome_Subseq
1
13
geeksforgeeks
Length of maximum palindrome Subsequence is 5
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ g++ Max_Palinrome_Subseq2.cpp -o Max_Palinrome_Subseq2
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_C
ode$ ./Max_Palinrome_Subseq2
1
13
geeksfoegeeks
Length of maximum palindrome Subsequence is 5
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$ ./Max_Palinrome_Subseq2
2
9
BBABCBCAB
Length of maximum palindrome Subsequence is 7
13
geeksforgeeks
Length of maximum palindrome Subsequence is 5
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$
```

Recursion Tree (Overlapping Subproblems)



Top Down (Memoization)

```cpp
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int length[MAX][MAX];
int SubSeq(char seq[], int i, int j)
{
  if(length[i][j]!=-1)
    return length[i][j];
  else
  {
    if(i==j)
    length[i][j]=1;
   else if(seq[i]==seq[j] && i==(j-1))
   {
      length[i][j]=2;
   }
   else if(seq[i]==seq[j])
    length[i][j] = SubSeq(seq,i+1,j-1)+2;
   else
    length[i][j] = max(SubSeq(seq,i+1,j), SubSeq(seq, i, j-1));
  }

   return length[i][j];
}

int main()
{
  int t;
  cin>>t;
  for(int i=0;i<t;i++)
  {

        int n;
        cin>>n;
    char seq[n];
        for(int j=0;j<n;j++)
        {
          cin>>seq[j];
        }
        for(int j=0;j<n;j++)
        {
          for(int k=0;k<n;k++)
            length[j][k]=-1;

        }
        cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,0,n-1)<<"\n";
  }
}
```

Bottom Up (Tebulation)

Matrix_Chain.cpp | Max_Palinrome_Subseq.cpp | Max_Palinrome_Subseq2.cpp | Max_Palinrome_Subseq3.cpp

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;

int SubSeq(char seq[], int n)
{
    int length[n][n];
    for(int i=0;i<n;i++)
        length[i][i]=1;
    for(int k=2;k<=n;k++)
    {
        for(int i=0;i<=(n-k);i++)
        {
            int j = k+i-1;
            if(k==2 && seq[i]==seq[j])
            {
                length[i][j]=2;
            }
            else
            {
                if(seq[i]==seq[j])
                {
                    length[i][j] = 2+length[i+1][j-1];
                }
                else
                {
                    length[i][j]=max(length[i][j-1],length[i+1][j]);
                }

            }
        }

```

Matrix Chain Multiplication

Problem : We are given a sequence(chain) $(A_1, A_2, ..., A_n)$ of n matrices to be multiplied. Our work is to find the most efficient way to multiply these matrices together.

Since matrix multiplication is associative, So we have many option to multiply a chain of matrices. **Ex.** Let the chain of matrices is (A_1, A_2, A_3, A_4) and we need to find $A_1.A_2.A_3.A_4$ There are multiple way to find this.

1. $((A_1.A_2).(A_3.A_4))$
2. $((A_1.(A_2.A_3)).A_4))$
3. $(A_1.((A_2.A_3).A_4))$
4. $(((A_1.A_2).A_3).A_4)$
5. $(A_1.(A_2.(A_3.A_4)))$

Example : $A_1 = 40*20, A_2 = 20*30, A_3 = 30*10, A_4 = 10*30$

1. $((A_1.A_2).(A_3.A_4)) = (40*20*30) + (30*10*30) + (40*30*30) = 70200$
2. $((A_1.(A_2.A_3)).A_4)) = 20*30*10 + 40*20*10 + 40*10*30 = 26000$
3. $(A_1.((A_2.A_3).A_4)) = 20*30*10 + 20*10*30 + 40*20*30 = 36000$
4. $(((A_1.A_2).A_3).A_4) = 40*20*30 + 40*30*10 + 40*10*30 = 48000$
5. $(A_1.(A_2.(A_3.A_4))) = 30*10*30 + 20*30*30 + 40*20*30 = 51000$

$((A_1.A_2).(A_3.A_4)) = (40*20*30) + (30*10*30) + (40*30*30)$

$$(A_1.A_2) : \text{Cost} = (40*20*30), \text{ Matrix Size} = 40*30$$
$$(A_3.A_4) : \text{Cost} = (30*10*30), \text{ Matrix Size} = 30*30$$
$$((A_1.A_2).(A_3.A_4)) : \text{Cost} = 40*30*30, \text{ Matrix Size} = 40*30$$
$$\text{Total Cost} = (40*20*30) + (30*10*30) + (40*30*30) = 70200$$
$$\text{Size of Resultant Matrix} = 40*30$$

Dynamic Approach :

Approach towards the solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value.

Suppose $p[n+1]$ is an array contain the size of each matrix. i.e

$$\text{size of } A_i = p[i-1] * p[i], \forall 1 \leq i \leq n$$

Let m[i,j] be the minimum number scalar multiplication needed to compute the multiplication of matrix A_i to A_j(i.e. $A_i.A_{i+1}.A_{i+2}.....A_j$).

So m[1,n] would be the lowest cost to compute the multiplication of matrix A_1 to A_n(i.e. $A_1.A_2.A_3.....A_n$).

A recursive solution :

We can split the product the product $A_i.A_{i+1}.A_{i+2}.....A_j$ between A_k and A_{k+1} where $i \leq k < j$ then m[i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$, plus the cost of multiplying these two matrices together.

size of $A_{i....k}$ = p[i-1]*p[k] and size of $A_{k+1.....j}$ = p[k]*p[j]

then multiplication cost of $A_{i....k} * A_{k+1....j} = p[i-1] * p[k] * p[j]$

$A_{i....k}$ = minimum cost to multiply the matrices $(A_i.A_{i+1}.A_{i+2}....A_k) = m[i,k]$

$A_{k+1....j}$ = minimum cost to multiply the matrices $(A_{k+1}.A_{k+2}....A_j) = m[k+1,j]$

$$m[i,j] = m[i,k] + m[k+1,j] + p[i-1] * p[k] * p[j]$$

A recursive solution :

m[i,j] = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]

Possible value of k is $j - i$ i.e $k = i, i + 1, i + 2, ..., j - 1$
We need to check them all values of k to find the best(minimum cost i.e optimal solution).

Recursive Relation :

$$m[i, j] = \begin{cases} 0, & \text{if } i = j \\ min(m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]), & \forall i \leq k < j \end{cases}$$

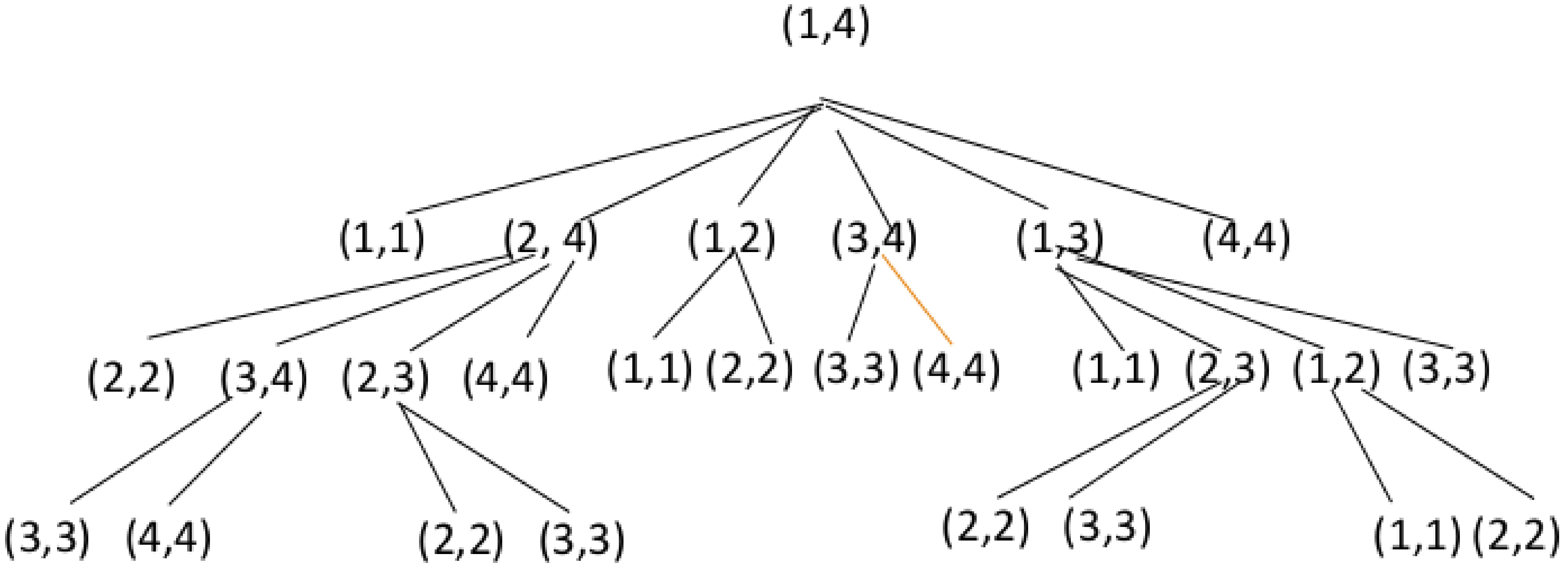
Recursive Code

```cpp
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
    return (x>y)?y:x;
}
int Matrix_Chain(int p[], int i,int j)
{
    int mult_cost = INT_MAX;
    if(i==j)
        return 0;
    else
    {
      for(int k=i;k<j;k++)
      {
        mult_cost = min(mult_cost,
                    Matrix_Chain(p,i,k)+Matrix_Chain(p,k+1,j)+p[i-1]*p[k]*p[j]);
      }
    }
    return mult_cost;
}
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        int p[n+1];
        for(int j=0;j<=n;j++)
        {
            cin>>p[j];
        }

    cout<<"Minimum cost for matrix multiplication is :"<<" "<<Matrix_Chain(p,1,
    }
}
```

Output

```
Minimum cost for matrix multiplication is : 15125
4
40 20 30 10 30
Minimum cost for matrix multiplication is : 26000
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$ ./Matrix_Chain_2
2
4
10 20 30 40 30
Minimum cost for matrix multiplication is : 30000
2
10 20 30
Minimum cost for matrix multiplication is : 6000
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$ g++ Matrix_Chain_3.cpp -o Matrix_Chain_3
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$ ./Matrix_Chain_3
4
6
30 35 15 5 10 20 25
Minimum cost for matrix multiplication is : 15125
4
40 20 30 10 30
Minimum cost for matrix multiplication is : 26000
4
10 20 30 40 30
Minimum cost for matrix multiplication is : 30000
2
10 20 30
Minimum cost for matrix multiplication is : 6000
rajkishor@rajkishor-HP-Pavilion-15-Notebook-PC:~/Documents/java_code/Unacademy_Code$ ./Matrix_Chain
4
6
30 35 15 5 10 20 25
Minimum cost for matrix multiplication is : 15125
4
40 20 30 10 30
Minimum cost for matrix multiplication is : 26000
4
```

Recursion Tree (Overlapping Subproblems)



Top Down (Memoization)

```cpp
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int mult[MAX][MAX];
int min(int x, int y)
{
    return (x>y)?y:x;
}
int Matrix_Chain(int p[], int i,int j)
{
    if(mult[i][j]!=-1)
        return mult[i][j];
    else
    {
        int mult_cost = INT_MAX;
        if(i==j)
          mult[i][j]=0;
        else
         {
          for(int k=i;k<j;k++)
           {
              mult_cost = min(mult_cost,
                 Matrix_Chain(p,i,k)+Matrix_Chain(p,k+1,j)+p[i-1]*p[k]*p[j]);
           }
           mult[i][j]=mult_cost;
         }

         return mult[i][j];
    }
    //return mult_cost;
}
```

```cpp
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        int p[n+1];
        for(int j=0;j<=n;j++)
        {
            cin>>p[j];
        }
        for(int j=0;j<MAX;j++)
        {
            for(int k=0;k<MAX;k++)
            {
                mult[j][k]=-1;
            }
        }

    cout<<"Minimum cost for matrix multiplication is :"<<" "<<Matrix_Chain(p,1,n)<<"\n";
    }
}
```

Bottom Up (Tebulation)

```cpp
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
    return (x>y)?y:x;
}
int Matrix_Chain(int p[], int n)
{
    int mult_cost[n][n];
    for(int i=1;i<n;i++)
    {
        mult_cost[i][i]=0;
    }
    for(int l=2;l<n;l++)
    {
        for(int i=1;i<(n-l+1);i++)
        {
            int j=i+l-1;
            mult_cost[i][j] = INT_MAX;
            for(int k=i;k<j;k++)
            {
                mult_cost[i][j] = min(mult_cost[i][j],
                    mult_cost[i][k]+mult_cost[k+1][j]+p[i-1]*p[k]*p[j]);
            }
        }
    }
    return mult_cost[1][n-1];
}
```

```cpp
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        int p[n+1];
        for(int j=0;j<=n;j++)
        {
            cin>>p[j];
        }

    cout<<"Minimum cost for matrix multiplication is :"<<" "<<Matrix_Chain(p,n+1)<<"\n";
    }
}
```

Boolean Parenthesization Problem

Problem:

Given a boolean expression with following symbols and operators :

Symbols : T for True and F for False
Operators : Boolean AND, Boolean OR , Boolean XOR.

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Ex. Symbols = (T,T,F,T), Operator = (OR,AND,XOR);
Expression = (T OR T AND F XOR T)

- ((T OR T) AND (F XOR T)) = T , (T OR ((T AND F) XOR T)) = T
  (T OR (T AND (F XOR T))) = T , (((T OR T) AND F) XOR T) = T.
- ((T OR (T AND F)) XOR T) = F

Total Expressions = 5, True Expressions = 4, False Expressions = 1.

Solution Approach :

Approach towards the solution is to place parenthesis at all possible places and evaluate the expression.
Let the Expression $= A_1\ X_1\ A_2\ X_2\ ......\ A_{n-1}\ X_{n-1}\ A_n$

$$A_i \in (T, F) \text{ and } X_i \in (AND, OR, XOR)$$

Total(i,j) : Total number of ways to paranthesize the given boolean expression which evaluates to either true or false.
T(i,j) : Total number of ways to paranthesize the given boolean expression which evaluates to true only.
F(i,j) : Total number of ways to paranthesize the given boolean expression which evaluates to false only.

Relation among Total(i,j), T(i,j) and F(i,j)

$$\textbf{Total(i,j)} = \textbf{T(i,j)} + \textbf{F(i,j)}$$

$$T(i,j) = \begin{cases} 1, & \text{if } i = j \quad symbol[i] = T \\ 0, & \text{if } i = j \quad symbol[i] = F \end{cases}$$

$$F(i,j) = \begin{cases} 1, & \text{if } i = j \text{ and } symbol[i] = F \\ 0, & \text{if } i = j \text{ and } symbol[i] = T \end{cases}$$

Subexpressions

Consider Subexpression between A_i to A_j

Subexpression $= A_i\ X_i\ A_{i+1}\ X_{i+1}\ \ldots\ldots\ A_{j-1}\ X_{j-1}\ A_j,$

For i=1 and j=n, it will be the actual given expression

Break the expression at any operator X_k $(i \le k < j)$.

Subexpression $= (A_i\ X_i\ A_{i+1}\ X_{i+1} \ldots X_{k-1}\ A_k)\ X_k\ (A_{k+1}\ X_{k+1} A_{k+2} A_{k+2} \ldots X_{j-1}\ A_j)$

Evaluation of Subexpressions

Subexpressions

1. $(A_i\ X_i\ A_{i+1}\ X_{i+1}\ ...X_{k-1}\ A_k)$
   T(i,k) = $T_{E1}, T_{E2}, ...$ F(i,k) = $F_{E1}, F_{E2}, ...$ Total(i,k) = $\sum T_{Ep} + \sum F_{Eq}$

2. $(A_{k+1}\ X_{k+1} A_{k+2} A_{k+2}...X_{j-1}\ A_j)$
   T(k+1,j) = $T_{E'1}, T_{E'2}, ...$ F(i,k) = $F_{E'1}, F_{E'2}, ...$ Total(k+1,j) = $\sum T_{E'p} + \sum F_{E'q}$

$$\begin{aligned}
\text{Total(i,j)} &= [(T_{E1}*(\textstyle\sum T_{E'p} + \sum F_{E'q}) + T_{E2}*(\sum T_{E'p} + \sum F_{E'q}) + ...] \\
&\quad + [(F_{E1}*(\textstyle\sum T_{E'p} + \sum F_{E'q}) + F_{E2}*(\sum T_{E'p} + \sum F_{E'q}) + ...] \\
&= [(T_{E1}+T_{E2}+....)+(F_{E1}+F_{E2}+....)] * [(\textstyle\sum T_{E'p} + \sum F_{E'q})] \\
&= [(\textstyle\sum T_{Ep} + \sum F_{Eq})] * [(\sum T_{E'p} + \sum F_{E'q})] \\
&= \text{Total(i,k)*Total(k+1,j)}
\end{aligned}$$

Recursive Relation

1. when X_k = AND

T(i,j) = $(T_{E1}*(\sum T_{E'p})) + (T_{E2}*(\sum T_{E'p})) + ... = \sum T_{Ep} * \sum T_{E'p}$ = T(i,k)*T(k+1,j)

F(i,j) = Total(i,j) - T(i,j) = Total(i,k)*Total(k+1,j) - T(i,k)*T(k+1,j)

2. when X_k = OR

F(i,j) = $(F_{E1}*(\sum F_{E'p})) + (F_{E2}*(\sum F_{E'p})) + ... = \sum F_{Ep} * \sum F_{E'p}$ = F(i,k)*F(k+1,j)

T(i,j) = Total(i,j) - F(i,j) = Total(i,k)*Total(k+1,j) - F(i,k)*F(k+1,j)

3. when X_k = XOR

T(i,j) = $[(T_{E1}*(\sum F_{E'p}))+(T_{E2}*(\sum F_{E'p}))+ ...] + [(F_{E1}*(\sum T_{E'p}))+(F_{E2}*(\sum T_{E'p}))+ ...]$

$= [\sum T_{Ep} * \sum F_{E'p}] + [\sum F_{Ep} * \sum T_{E'p}]$

= T(i,k)*F(k+1,j) + F(i,k)*T(k+1,j)

Similarly, F(i,j) = T(i,k)*T(k+1,j) + F(i,k)*F(k+1,j)

Recursive Relation

For $i \leq k < j$

Case 1 : when X_k = AND

$\mathbf{T(i,j)} = \sum_{k=i}^{j-1}$ **T(i,k)*T(k+1,j) and**

$\mathbf{F(i,j)} = \sum_{k=i}^{j-1}$ **Total(i,k)*Total(k+1,j) - T(i,k)*T(k+1,j)**

Case 2 : when X_k = OR

$\mathbf{F(i,j)} = \sum_{k=i}^{j-1}$ **F(i,k)*F(k+1,j) and**

$\mathbf{T(i,j)} = \sum_{k=i}^{j-1}$ **Total(i,k)*Total(k+1,j) - F(i,k)*F(k+1,j)**

Case 3 : when X_k = XOR

$\mathbf{T(i,j)} = \sum_{k=i}^{j-1}$ **T(i,k)*F(k+1,j) + F(i,k)*T(k+1,j) and**

$\mathbf{F(i,j)} = \sum_{k=i}^{j-1}$ **T(i,k)*T(k+1,j) + F(i,k)*F(k+1,j)**

Implemented Code :

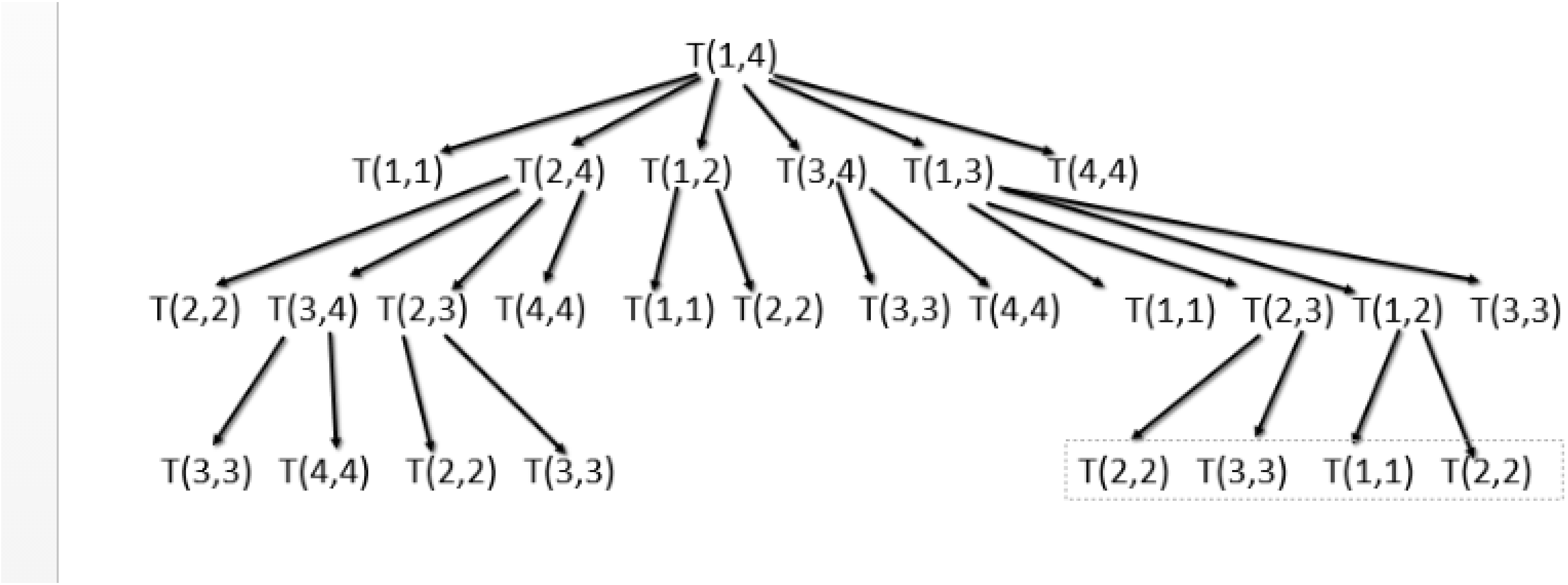
Boo' Boolean_Paranthesis4.cpp × Boolean_Paranthesis.java × Boolean_Paranthesis2.java × anthesis3.java

```java
import java.util.*;
class Boolean_Paranthesis
{
    public static int TOTAL(String symbol[], String ope[], int i, int j)
    {
        return TRUE_EXP(symbol,ope,i,j)+FALSE_EXP(symbol,ope,i,j);
    }
    public static int FALSE_EXP(String symbol[], String ope[], int i, int j)
    {
        if(i==j)
        {
            return (symbol[i].equals("F"))?1:0;
        }
        int true_count =0;
        for(int k=i;k<j;k++)
        {
            if(ope[k].equals("&"))
                true_count += TOTAL(symbol,ope,i,k)*TOTAL(symbol,ope,k+1,j) - TRUE_EXP(symbol,ope,i,k)*TRUE_EX
            else if(ope[k].equals("|"))
                true_count += FALSE_EXP(symbol,ope,i,k)*FALSE_EXP(symbol,ope,k+1,j);
            else
                true_count += TRUE_EXP(symbol,ope,i,k)*TRUE_EXP(symbol,ope,k+1,j) + FALSE_EXP(symbol,ope,i,k)*

        }
        return true_count;
    }
```

```java
}
public static int TRUE_EXP(String symbol[], String ope[], int i, int j)
{
    if(i==j)
    {
        return (symbol[i].equals("T"))?1:0;
    }
    int true_count =0;
    for(int k=i;k<j;k++)
    {
        if(ope[k].equals("&"))
            true_count += TRUE_EXP(symbol,ope,i,k)*TRUE_EXP(symbol,ope,k+1,j);
        else if(ope[k].equals("|"))
            true_count += TOTAL(symbol,ope,i,k)*TOTAL(symbol,ope,k+1,j) - FALSE_EXP(symbol,ope,i,k)*FALSE_E
        else
            true_count += TRUE_EXP(symbol,ope,i,k)*FALSE_EXP(symbol,ope,k+1,j) + TRUE_EXP(symbol,ope,k+1,j)

    }
    return true_count;
}
public static void main(String args[])
```

```java
    }
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);

        int t = input.nextInt();
        for(int i=0;i<t;i++)
        {

            int n = input.nextInt();
            String symbol[] = new String[n];
            String ope[] = new String[n];
            for(int j=0;j<n;j++)
            {
                symbol[j] = input.next();
            }
            for(int j=0;j<(n-1);j++)
            {
                ope[j] = input.next();
            }
            System.out.println("No of ways : " + TRUE_EXP(symbol,ope,0,n-1));

        }
    }

}
```

Recursion Tree Diagram



- Overlapping Subproblems
- Unnecessary Multiple Computations

Top Down or Memoization

Boolean_Paranthes Boolean_Paranthesis.java × Boolean_Paranthesis2.java × Boolean_Paranthesis3.java ×

```java
import java.util.*;
class Boolean_Paranthesis3
{
    static int T[][] = new int[200][200];
    static int F[][] = new int[200][200];
    public static int TOTAL(String symbol[], String ope[], int i, int j)
    {
        return TRUE_EXP(symbol,ope,i,j)+FALSE_EXP(symbol,ope,i,j);
    }
    public static int FALSE_EXP(String symbol[], String ope[], int i, int j)
    {
        if(F[i][j]!=-1)
        {
          return F[i][j];
        }
        if(i==j)
        {
            if(symbol[i].equals("F"))
                F[i][j]=1;
            else
                F[i][j]=0;
            return F[i][j];
        }
        int true_count =0;
        for(int k=i;k<j;k++)
        {
```

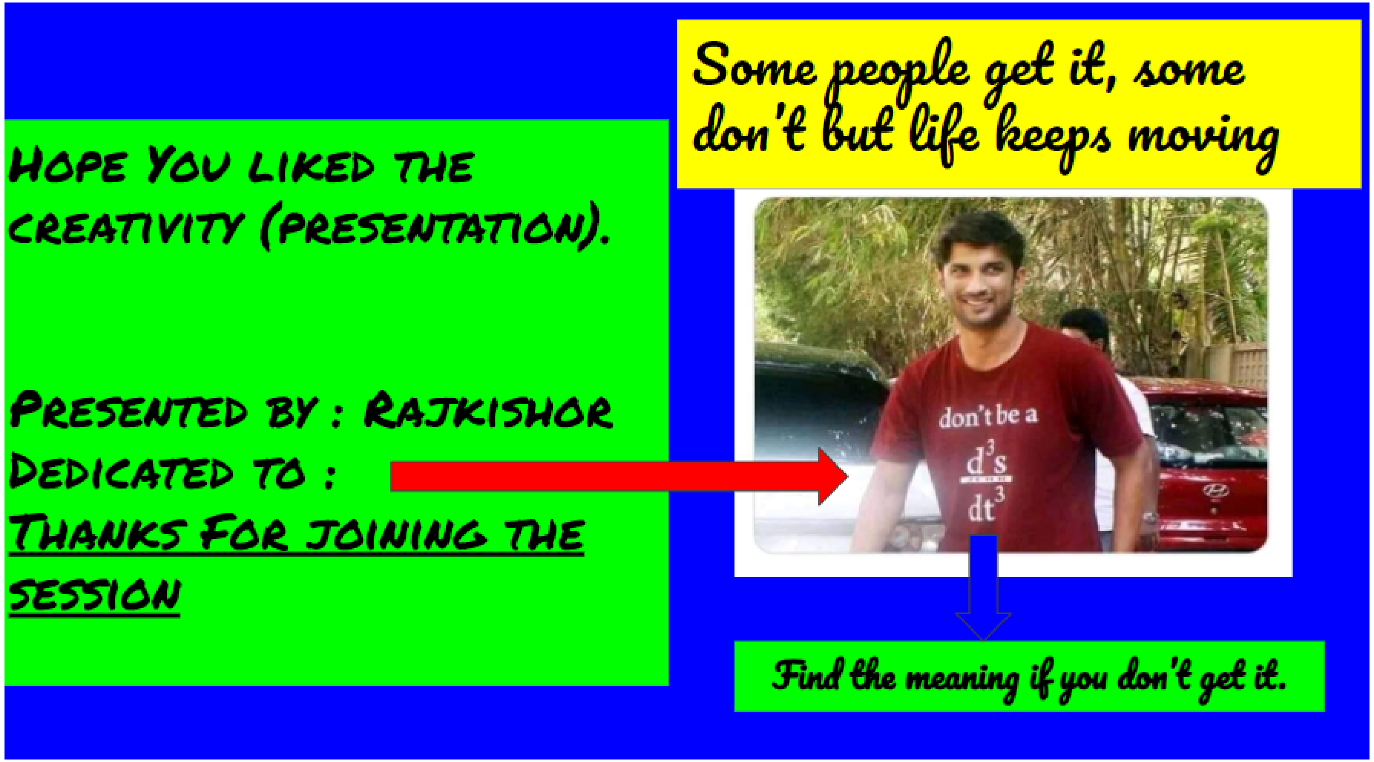
```java
public static int TRUE_EXP(String symbol[], String ope[], int i, int j)
{
    if(T[i][j]!=-1)
    {
        return T[i][j];
    }
    if(i==j)
    {
        if(symbol[i].equals("T"))
            T[i][j]=1;
        else
            T[i][j]=0;
        return T[i][j];
    }
    int true_count =0;
    for(int k=i;k<j;k++)
    {
        if(ope[k].equals("&"))
          true_count += TRUE_EXP(symbol,ope,i,k)*TRUE_EXP(symbol,ope,k+1,j);
        else if(ope[k].equals("|"))
          true_count += TOTAL(symbol,ope,i,k)*TOTAL(symbol,ope,k+1,j) - FALSE_EXP(symbol,ope,i,k)*FALSE_E
        else
          true_count += TRUE_EXP(symbol,ope,i,k)*FALSE_EXP(symbol,ope,k+1,j) + TRUE_EXP(symbol,ope,k+1,j)

    }
    T[i][j]=true_count;
    return T[i][j];
}
```

Top Down or Memoization

Boolean_Paranthe Boolean_Paranthesis.java × Boolean_Paranthesis2.java × Boolean_Paranthesis3.java ×

```java
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);

        int t = input.nextInt();
        for(int i=0;i<t;i++)
        {

            int n = input.nextInt();
            String symbol[] = new String[n];
            String ope[] = new String[n];
            for(int j=0;j<n;j++)
            {
                symbol[j] = input.next();
            }
            for(int j=0;j<(n-1);j++)
            {
                ope[j] = input.next();
            }
            for(int j=0;j<200;j++)
            {
                for(int k=0;k<200;k++)
                {
                    T[j][k]=-1;
                    F[j][k]=-1;
                }
            }
            System.out.println("No of ways : " + TRUE_EXP(symbol,ope,0,n-1));
```

End Of Presentation



(1).png