Algorithms

13. DP

Handwritten by pankaj kumar

15. Dynamic Programming.

Date
Page No. 122.

① Introduction :-

- Dynamic programming (commonly referred to as DP) is an algorithmic technique for solving a problem by recursively breaking it down into simpler subproblems and using the fact that the optimal solution to the overall problem depends upon the optimal solution to it's individual subproblem
- The technique was developed by Richard Bellman in the 1950s.
- DP algorithm solve each subproblem just once and then remember its answer, thereby avoiding re-computation of the answer for similar subproblem every time.
- It is the most powerful design technique for solving optimization related problems.

* Characteristics of Dynamic Programming.

We can apply DP technique to those problems that exhibit the below 2 characteristics:

① Optimal Substructures:

- Any problem is said to be having optimal substructure property if its overall optimal solution can be evaluated from the optimal soln of its subproblems.

② Overlapping Subproblem:

- Subproblems are basically the smaller versions of an original problem. Any problem is said to have overlapping subproblem if calculating its solution involves solving the same subproblem multiple times.
- In DP technique we store the answer of some problem and use it whenever we have same subproblem occur.

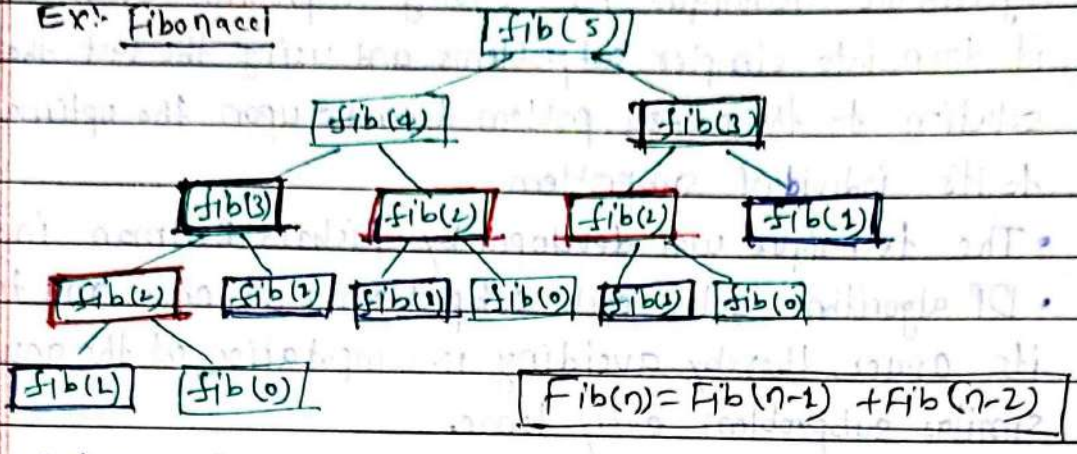
* Dynamic Programming Methods

We can use any one of these technique to solve a problem in optimised manner.

① Top Down Approach (Memoization):

- Top Down Approach is the method where we solve a bigger problem by recursively finding the solution to smaller sub-problems

Ex:- Fibonacci



- Whenever we solve a smaller subproblem, we remember (cache) its result so that we don't solve it repeatedly if it's called many times. Instead of solving repeatedly, we can just return the cached result.
- This method of remembering the solution of already solved subproblem is called memoization.

② Bottom Up Approach (Tabulation):

- As the name indicates, bottom up is the opposite of the top-down approach which avoids recursion.
- Here, we solve the problem "bottom-up" way, i.e. by solving all the related subproblems first. This is typically done by populating an n-dimensional table.
- Depending on the result in the table, the solution to the original problem is then computed. This approach is therefore called as "Tabulation".

* Application (standard Problem)

1. Largest common subsequences (LCS)
2. Longest increasing subsequences (LIS)
3. Knapsack problem.
4. Kadane's Algorithm
5. Matrix chain multiplication
6. DP on trees
7. DP on Grid

(2) Nth Fibonacci :-

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

① Recursive solution without memoization.

```
int Fib(int n){
    if (n<=1) return n;   //Fib(0)=0; Fib(1)=1;
    return Fib(n-1)+Fib(n-2);
}
```

Time: exponential in terms of n bcz some terms are evaluated again & again.

② With Memoization

```
int memo[100]={0};
int Fib(int n){
    if (n<=1) return n;
    if (memo[n]!=0) return memo[n];
    memo[n] = Fib(n-1)+Fib(n-2);
    return memo[n];
}
```

Space: since we are using recursion to solve this, we also end up using stack memory as part of recursion overhead which is also O(n). so overall space complexity is O(n)+O(n) = 2O(n) = O(n).

Time: ∵ Fib(n) = Fib(n-1) + Fib(n-2)

First Fib(n-1) will called when it will return Fib(n-2) is already calculated so it will used value of Fib(n-2) from memoization. O(1)

∴ Time will be. $T(n) = T(n-1) + c$

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

$= T(n-3) + 3c$

$= T(n-k) + kc$

$= T(0) + n*c = 1 + n*c = O(n)$

(3) Tabulation method:

- We already know $\boxed{Fib(n) = Fib(n-1) + Fib(n-2)}$
- Based on the above relation, we calculate the results of smaller subproblems first and then build the table.

```
int fib(int n){
    int dp[n+1];
    int i;
    dp[0]=0; dp[1]=1;
    for(i=2; i<=n; i++)
        dp[i]=dp[i-1]+dp[i-2];
    return dp[n];
}
```

Space & Time : O(n)

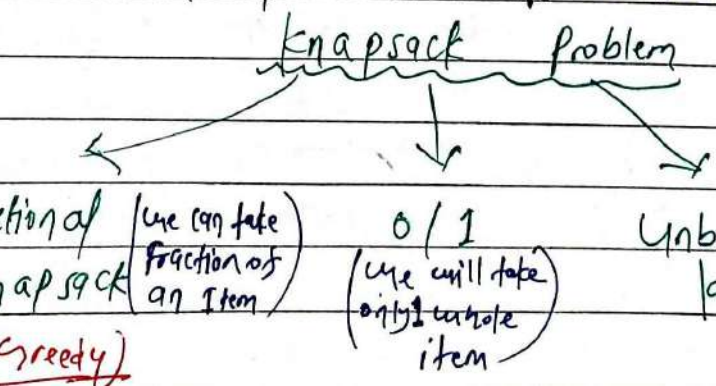
- We can reduce space complexity from O(N) to O(1) by just using 2 variables. This is left as an assignment to the reader.

---

(3) 0-1 knapsack Problem :-

1. Subset Sum
2. Equal Sum Partition
3. Count of Subset Sum
4. minimum Subset Sum Diff.
5. Target Sum
6. # of subset with given diff.

} Related Problem



• Given a knapsack / Bag with W weight capacity and a list of N items with given v_i value and w_i weight. Put these items in the knapsack in order to maximize the value of all the placed items without exceeding the limit of knapsack.
• The item cannot be break you can either select it fully (1) or don't select it (0).

*Ex:- W = 40
value = [10|20|30|40]
weights = [30|10|40|20]

Output : Item2 + Item4 = 60

* Method 1 (Using Bruteforce Recursion):
- we will create all the subset of items with total weight less than that of the given capacity W. From result we will return the subset with maximum value.
- For every element we can,
• either select it or, • ignore and move forward.
- if we select an item then its value will be added to our current value and weight will be subtracted from available

```
int knapsack01(int W1, int N, vector<int> &v, vector<int> &w)
{
    if (N==0 || W1==0)
        return 0;
    if (w[N] <= W1){
        return max(v[n] + knapsack01(W1-w[N], N-1,
        v,w), knapsack01(W1, N-1, v, w));
    }
    else if (w[N] > W1)
        return knapsack01(W1, N-1, v, w);
}
```

Time: $O(2^n)$

*Method 2 (Using dynamic Programming):
in the above approach we can observe that we are calling

recursion for same sub problems again & again thus resulting in overlapping subproblems thus we can make use of DP.

* using memoization

```
int knapsackrecur(int W1, int N, vector<int>& v, vector<int> &w,
                  vector<vector<int>> & dp){
    if (N==0 || W1==0)
        return 0;
    if (dp[N][W1] != -1) return dp[N][W1];
    if (W[N] <= W1)
      return dp[N][W1] = max(v[N]+knapsackrecur(W1-W[N],
        N-1, V, W, dp), knapsackrecur(W1, N-1, V, W, dp));
    else if(W[N]>W1)
    return dp[N][W1] = knapsackrecur(W1, N-1, V, W, dp);
}

int knapsack01(int W1, int N, vector<int> & V, vector<int>
    &W){
    vector<vector<int>> dp(N+1, vector<int> (W1+1, -1));
    return knapsackrecur(W1, N-1, V, W, dp);
}
```

Time O(N*W1)
space O(N*W1)

* Using iterative DP

```
int knapsack01(int W1, vector<int>& v, vector<int> &w){
    int DP[N+1][W1+1];
    for(int i=0; i<N+1; i++) DP[i][0]=0;
    for(int i=0; i<W1+1; i++) DP[0][i]=0;
```

Base case of Recursion

According to base case when N=0 & W=0, assume 0

```
for(int i=1; i<N+1; i++){
    for(int j=1; j<W1+1; j++){
        if(W[i-1] <= j){
            DP[i][j] = max(V[i-1] + DP[i-1][j-W[i-1]],
                           DP[i-1][j]);
        }
        else{
            DP[i][j] = DP[i-1][j];
        }
    }
}
return DP[N][W1];
}
```

Same as Recursive call

Time: $O(N*W_1)$
Space: $O(N*W_1)$

**** * Space Optimized :

• here result for the nth element we just need the result for the (n-1)th elements. ∴ we will only store the (n-1)th elements result

```
int knapsack01(int W1, int N, vector<int> &V, vector<int> &W){
    int DP[W+1];
    for(int i=0; i<W1+1; i++) DP[i]=0;
    for(int i=1; i<N+1; i++){
        for(int j=W1; j>=W[i-1]; j--){
            DP[j] = max(V[i-1] + DP[j-W[i-1]],
                        DP[j]);
        }
    }
    return DP[W1];
}
```

Time: $O(N*W_1)$
Space: $O(W_1)$

④ Subset Sum :

Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Ex:-

set[] = {3, 34, 4, 12, 5, 2}, sum = 9.

O/P = True

Subset is {4, 5}

* Method 1 (Recursion)

```
bool isSubsetSum(int set[], int n, int sum){
    if (sum == 0)
        return true;
    if (n == 0)
        return false;
    if (set[n-1] > sum)
        return isSubsetSum(set, n-1, sum);
    else
    return isSubsetSum(set, n-1, sum) ||
        isSubsetSum(set, n-1, sum-set[n-1]);
}
```

Time = exponential (2^n)

* Method 2 (memoization technique)

```
int tab[2000][2000]; (Global declaration)
int isSubsetSum(int set[], int n, int sum){
    if (sum == 0)
        return 1;
    if (n <= 0)
        return 0;
    if (tab[n-1][sum] != -1)
        return tab[n-1][sum];
```

[Note:- We can optimize space complexity of tabulation method as previous (0/1 knapsack) space optimization technique]


```
if (set[n-1] > sum)
    return tab[n-1][sum] = isSubsetSum(set, n-1, sum);
else
    return tab[n-1][sum] = isSubsetSum(set, n-1, sum)
                        || isSubsetSum(set, n-1, sum-set[n-1]);
}
```

Time: O(sum*n)
Space: O(sum*n)

* Method 3 (Tabulation Method)

set[] = {3, 4, 5, 2}, target = 6

	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
3	T	F	F	T	F	F	F
4	T	F	F	T	T	F	F
5	T	F	F	T	T	T	F
2	T	F	T	T	T	T	T

- In tabulation method we directly fill the memoization table and the last index value is our answer.

```
bool isSubsetSum(int set[], int n, int sum){
    bool subset[n+1][sum+1];
    for(int i=0; i<=n; i++) subset[i][0] = true;
    for(int i=1; i<=sum; i++) subset[0][i] = false;
    for(int i=1; i<=n; i++)
        for(int j=1; j<=sum; j++){
            if(j<set[i-1])
                subset[i][j] = subset[i-1][j];
            else
                subset[i][j] = subset[i-1][j] ||
                               subset[i-1][j-set[i-1]];
        }
    return subset[n][sum];
}
```

Time O(sum*n)
Space O(sum*n)

(5) Partition Sum Problem / Equal sum partition :-

Partition problem is to determine whether a given set can be partitioned into two subset such that the sum of elements in both subsets is the same.

Ex: arr[] = {1, 5, 11, 5}

o/p: true

The array partitioned as {1, 5, 5} and {11}

Approach :-

(1) Calculate sum of array. if sum is odd, there can not be two subset with equal sum, so return false

(2) if sum of array element is even, calculate sum/2 and find a subset of array with sum equal to sum/2

second step can be solve using previous problem Subset Sum.

Code :-

```
bool findPartition (int arr[], int n) {
    int sum = 0;
    for (int i=0; i<n; i++) sum += arr[i];
    if (sum % 2 != 0) return false;
    else return isSubsetSum (arr, n, sum/2);
}
```

Note :- bool isSubsetSum(int arr[], int n, int sum);

⇒ It is same as previous problem (subset sum)

Time O(sum * n)

Space O(sum * n) / space can be optimized

(6) Count of subsets with sum equal to X.

Given an array arr[] of length N and an integer X, the task to find the number of subsets with a sum equal to X.

Ex:- I/P: arr[] = {1, 2, 3, 3}, X = 6

O/P: 3

Possible subset are {1, 2, 3}, {1, 2, 3} and {3, 3}

* Approach:-

We use same approach of finding the subset sum problem, but here code variation will be instead of returning true & false we will return 1 & 0 (in base case). and instead of || (or condition) we use '+' then code will be.

* Without memoization

```
int SubsetSum (int arr[], int n, int x){
        if (x == 0) return 1;
        if (n == 0) return 0;
        if (arr[n-1] > x) return SubsetSum(arr, n-1, x)
        else
            return SubsetSum(arr, n-1, x) + SubsetSum(arr, n-1,
                                                      x-arr[n-1]);
}
```

Time: $O(2^n)$

Note:- We can use the same approach with memoization & tabulation as subset sum problem (true → 1, false → 0, || → +)

* With memoization :-

```
int tab[2000][2000];
int SubsetSum(int arr[], int n, int x){
        if (x == 0) return 1;
        if (n == 0) return 0;
        if (tab[n-1][x] != -1) return tab[n-1][x];
        if (arr[n-1] > x)
            return tab[n-1][x] = SubsetSum(arr, n-1, x)
```

Time O(x*n)
Space O(x*n)

else
return tab[n-1][x] = Subsetsum(arr, n-1, x) +
Subsetsum(arr, n-1, x-arr[n-1]);

* tabulation method

```
int subsetsum(int a[], int n, int x){
    int tab[n+1][x+1];
    tab[0][0] = 1; // First value of matrix
    for(int i=1; i<=x; i++) tab[0][i] = 0;

    for(int i=1; i<=n; i++)
        for(int j=0; j<=x; j++){
            if (a[i-1] > j)
                tab[i][j] = tab[i-1][j];
            else
                tab[i][j] = tab[i-1][j] + tab[i-1][j-a[i-1]];
        }

    return tab[n][x];
}
```

Time O(x*n)
Space O(x*n)

* Space optimized (x) (n)

```
int subsetsum(int arr[], int val, int n){
    int count = 0;
    vector<int> PresentState(val+1, 0), LastState(val+1, 0);
    PrerentState[0] = LastState[0] = 1;
    if (arr[0] <= val) LastState[arr[0]] = 1;
    for(int i=1; i<n; i++){
        for(int j=0; j<=val; j++)
            presentstate[j] = (j >= arr[i]) ? LastState[j-arr[i]]
                              : 0) + LastState[j];
        LastState = PresentState;
    }
    return PresentState[val];
}
```

Time: O(x*n)
space: O(x)

Date

Page No. 135.

⑦ Minimum Subset Sum Difference :

- Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference b/w their sums is minimum.

Ex:- I/P: arr[] = {1, 6, 11, 5}

O/P: 1

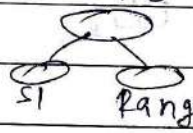
Explanation: {1, 5, 6}, sum of subset1 = 12
{11}, sum of subset2 = 11
] difference is '1'

Equal sum:- S1 - S2 = 0

A/c Question.

abs (S1 - S2) = min

since the range can be range = S1 + S2 ∴ S2 = range - S1



Now, abs (Range - S1 - S1) minimize = abs (Range - 2S1)

* Recursion

* Now we have to find the subset sum (S1) which is just less than or equal to (total sum/2) i.e maxm sum in the range with sum/2. after then minimum dist. will be (Range - 2S1)

* Tabulation :

	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3	T	T	T	T	F	F	F	T	T	T	T

we will push the last row in vector till half array (i.e half sum) and then check for

minm = min(minm, Range - 2 * v[i])

```
int minDiff (int arr[], int n) {
    int sum=0;
    for(int i=0; i<n; i++) sum += arr[i];
    bool dp[n+1][sum+1];
    for (int i=0; i<n+1; i++) {
    for (int j=0; j<sum+1; j++) {
        if (i==0) dp[i][j]= false;
        if (j==0) dp[i][j] = true;
    }
    }
    for(int i=1; i<n+1; i++) {
        for(int j=1; j<sum+1; j++) {
            if (arr[i-1] <=j)
                dp[i][j] = dp[i-1][j-arr[i-1]] || dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j];
    } }
    int diff=INT_MAX;
    for(int j= sum/2; j>=0; j--) {
        if (dp[n][j]==true) {
            diff = min(sum-2*j, diff)
    } }
    return diff;
}
```

(8) Count the number of subset with given difference.

- We have given an array and a difference, we need to find how many subsets are there in the array such that the difference b/w the sum of the two subset is equal to the given difference.

* Approach:

Let's take sum1 and sum2 be the subset and sum be the total sum of the array then.

$$sum1 - sum2 = diff.$$
$$+ \quad sum1 + sum2 = sum$$
$$\therefore sum1 = \frac{sum + diff}{2}$$

- so, simply we can get the answer by finding the number of subset with given sum = sum1, which is same as subset sum problem.

---

(9) Target sum

assign symbols to make the sum of nums to target.

Ex:-

N = 5

A[] = {1, 1, 1, 1, 1}

target = 3

O/P = 5

-1+1+1+1+1=3

+1-1+1+1+1=3

+1+1-1+1+1=3

+1+1+1-1+1=3

+1+1+1+1-1=3

* Approach:-

- this problem is same as previous problem. here actually the target is the difference b/w two subset.
- so we can rewrite this as

$$sum1 = \frac{sum + target}{2}$$

- and count the number of subset with equal to sum = sum1.

Related problem to unbounded knapsack
(a) Rod cutting
(b) coin change 2
(c) coin change 2.

(10.) Unbounded knapsack (Repetition of item allowed):

I/P: W=100
val[]={1,30}
wt[]={1,50}

O/P: 100

There are many ways to fill knapsack.
1. 2 instances of 50 unit weight item.
2. 100 instances of 1 unit weight item
3. 1 instances of 50 unit weight item and 50 instances of 1 unit item

we get maxm value with option 2.

* Brute-Force: Recursive solution
- Try all possible combination of given item and choose one with maximum profits.
- The only difference from 0/1 knapsack is after including the item we recursively call to process all items (instead of remaining)

(w: weight, P: profit, C: capacity)

```
def unbounded_knapsack(w, P, C):
    item_index=0
    return unbounded_knapsack1(w, P, C, item_index)

def unbounded_knapsack1(w, P, C, idx):
    if (C<=0 or idx >= len(P[redacted])):
        return 0
    include, exclude = 0,0
    if (w[idx] <= C):
        include = P[idx]+unbounded_knapsack1(w, P, C-w[idx], idx)
    exclude=unbounded_knapsack1(w, P, C, idx+1)

    return max(include, exclude)
```

Time: $O(2^{N+C})$
Space: $O(N+C)$

* DP + Recursion + Memoization

```
def 4k(w, p, c):
    memory = [[None] * (c+1) for i in range(len(w))]
    idx = 0
    return 4k1(w, p, c, idx, memory)

def 4k1(w, p, c, idx, memory):
    if (c <= 0 or idx >= len(p)):
        return 0
    if (memory[idx][c]):
        return memory[idx][c]
    include, exclude = 0, 0
    if (w[idx] <= c):
        include = p[idx] + 4k1(w, p, c - w[idx], idx, memory)
    exclude = 4k1(w, p, c, idx+1, memory)
    memory[idx][c] = max(include, exclude)
    return memory[idx][c]
```

Time: O(N*C)
Space: O(N*C)

* DP + Iteration + Tabulation

- For every possible capacity 'c' (0 <= c <= capacity), we have two options.
  1. Exclude the item, In this case we will take whatever profit we get from the sub-array excluding this item table[index-1][c].
  2. Include the item, if weight is lesser than the capacity available, in this we get profit by this items plus profit from remaing capacity profit[idx] + table[index][c]
- Finally take maxm of both

Time: O(N*C)
Space: O(N*C)

```
def uk(wt, p, c):   # weights -> profits -> capacity
    n = len(p)
    table = [[0] * (capacity + 1) for i in range(n)]

    For i in range(n): table[i][0] = 0
    # when only 1 item is considered get the profits using
    # that item for every capacity 'c'
    For c in range (capacity + 1):
        table[0][c] = (c // weights[0]) * profit[0]

    For i in range(1, n):
        for c in rang(1, capacity + 1):
            if (weights[i] <= c):
                table[i][c] = max(table[i-1][c],
                                  profits[i] + table[i][c-weights[i]])
            else:
                table[i][c] = table[i-1][c]
    return table[i][c]
```

(11) Rod Cutting Problem :-

Given a rod of length n and a list of rod prices of length i, where $1 <= i <= n$, Find the optimal way to cut the rod into smaller rod to maximize profit.

Ex:-

length[] = [1, 2, 3, 4, 5, 6, 7, 8]

price[] = [1, 5, 8, 9, 10, 17, 17, 20]

Rod length = 4

Best:- cut the rod into two pieces of length two (2) to gain revenue of 5 + 5 = 10

*using the idea of unbounded knapsack.

```
int t[n][n];
int un_kp (int price[], int length[], int max_len, int n){
    if (n==0 || max_len==0) return 0;
    if (t[n][max_len] != -1) return t[n][max_len];
    if (length[n-1] <= max_len)
        return t[n][max_len] = max (price[n-1] + un_kp (price,
            length, max_len - length[n-1], n),
            un_kp (price, length, max_len, n-1));
    else
        return t[n][max_len] = un_kp (price, length, max_len, n-1);
}
```

Time: O(max_len * n)
Space: O(max_len * n)

*Iterative + DP

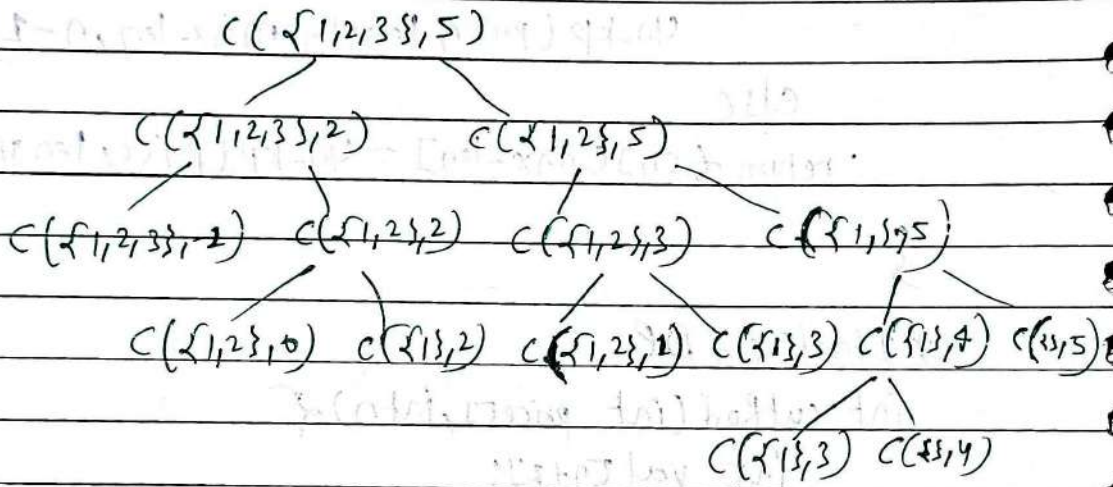
```
int cutRod (int price[], int n){
    int val[n+1];
    val[0]=0;
    int i,j;
    for (i=1; i<=n; i++){
        int max_val = INT_MIN;
        for (j=0; j<i; j++){
            max_val = max (max_val, price[j] + val[i-j-1]);
        val[i]=max_val;
        }
    }
    return val[n];
}
```

(13) Coin change I (maxm no. of ways):

I/P coin[] = [1|2|3] , coins are unlimited
sum: 5

O/P: 5
- 2+3=5
- 1+2+2=5
- 1+1+3=5
- 1+1+1+1+1=5
- 1+1+1+2=5

• Relation:-
unbounded knapsack, count of subset sum



* Recursion + memoization

required coin (size of coin array)

```
int coinchange(vector<int>& a, int n, int m, vector<vector<int>>
                                                    & dp) {
    if(n==0)  return dp[m][n]=1;
    if(m==0)  return 0;
    if(dp[m][n] != -1) return dp[m][n];
    if(a[m-1]<=n){
      //Either pick or not
      return dp[m][n]= coinchange(a, n-a[m-1], m, dp) +
                       coinchange(a, n, m-1, dp);
    }
    else
      return dp[m][n]= coinchange(a, n, m-1, dp);
}
```

Time: O(m*n)
Space: O(m*n)

* Top down DP: (unlimited coin list, no. of coin, size of coin array, sum of coin)

```
int count(int s[], int m, int n){
    int i, j, x, y;
    int table[n+1][m];
    for(i=0; i<m; i++) table[0][i]=1;
    for(i=1; i<n+1; i++){
        for(j=0; j<m; j++){
            x = (i-s[j]>=0)? table[i-s[j]][j] : 0;
            y = (j>=1)? table[i][j-1] : 0;
            table[i][j] = x+y;
        }
    }
    return table[n][m-1];
}
```

* Top down Space optimized:

```
int count(int s[], int m, int n){
    int table[n+1];
    memset(table, 0, sizeof(table));
    table[0]=1;
    for(int i=0; i<m; i++)
        for(int j=s[i]; j<=n; j++)
            table[j] += table[j-s[i]];
    return table[n];
}
```

(13) Coin change II (minm no. of coin):

tip: coin[]: [1|2|3]
sum: 5

op: 2 {2+3=5 (min no. of coin)}

* Recursive (without DP)

Time (O(2^n)),

coinsarray size target

```
int minCoins(int coins[], int m, int V){
    if (V==0) return 0;
    int res = INT_MAX;
    for(int i=0; i<m; i++){
        if (coins[i] <= V){
            int sub_res = minCoins(coins, m, V-coins[i]);
            if (sub_res != INT_MAX && sub_res+1 < res)
                res = sub_res + 1;
        }
    }
    return res;
}
```

* DP + Bottom-up + Iterative

```
int minCoins(int coins[], int m, int V){
    int table[V+1];
    table[0] = 0;
    for(int i=1; i<=V; i++)
        table[i] = INT_MAX;
    for(int i=1; i<=V; i++)
    {
        for(int j=0; j<m; j++)
            if (coins[j] <= i){
                int sub_res = table[i-coins[j]];
                if (sub_res != INT_MAX && sub_res+1 < table[i])
                    table[i] = sub_res + 1;
            }
    }
    if (table[V] == INT_MAX)  return -1;
    return table[V];
}
```

(14) Longest common subsequence :-

Based Problem

1. Print LCS
2. Shortest common super sequence
3. print SCS
4. minm no. of insertion & deletion
5. Largest Repeating subsequences
6. length of largest subsequences of 'a' which is a substring of 'b'
7. subsequences pattern matching
8. count how many times a appears subsequences in b
9. Largest palindromic subsequence.
10. Largest palindromic substring
11. Count of palindromic substring
12. minm no. of deletion in a string to make it a palindrome
13. minm no. of insertion in a string to make it a palindrome.

Longest Common Subsequence :

I/P: x: a b c d g h
y: a b e d f a

O/P: 4 (a b d h)

- we have given two string we have to find the longest common subsequence length.

process :- (choise diagram)

case1: if last character of string match then increare LCS answer by 1 and call LCS by reducing size by 1 of both string.

x: abcdgh
y: abedfh

LCS of → x: abcdg, y: abedf

case2: if last charact not match then.

x: abcdgh
y: abedfha

LCS of → x: abcdg, y: abedfha

LCS of → x: abcdgh, y: abedfh

① Recursion code n = length of x | m = length of y

```
int LCS(string x, string y, int n, int m){
    if(n==0 || m==0) return 0;
    if (x[n-1] == y[m-1])
        return 1+LCS(x,y,n-1,m-1);
    else
        return max(LCS(x,y,n-1,m), LCS(x,y,n,m-1));
}
```

// TC: O(2^n)

② memoization implementation

```
int dp[1001][1001];
int LCS (string x, string y, int n, int m){
    if(n==0 || m==0) dp[n][m]=0;
    if (dp[n][m] != -1) return dp[n][m];
    if (x[n-1] == y[m-1])
        dp[n][m] = 1+LCS(x,y,n-1,m-1);
    else
        dp[n][m] = max(LCS(x,y,n-1,m), LCS(x,y,n,m-1));
    return dp[n][m];
}
```

// TC: O(mn)

Note:- dp[m][n], is the length of LCS for strings x & y of length m & n.

③ tabulation implementation

```
int LCS (string x, string y, int n, int m){
    int dp[n+1][m+1];
    for(int i=0; i<=n; i++)
        for(int j=0; j<=m; j++)
            if(i==0||j==0)
                dp[i][j]=0;
```

{ base case of recursion when one of the string length is '0' it means length of LCS is zero. }

(when last character is same then LCS length is more than one from previous LCS)

```
for (int i=1; i<=n; i++)
    for (int j=1; j<=m; j++)
        if (x[i-1] == y[j-1])
            dp[i][j] = 1 + dp[i-1][j-1]
        else
            dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
return dp[n][m];
}
```

// TC: O(mn)

[lengths of LCS when length of string is n & m (length of original string)]

Note:- dp[i][j] = length of LCS when string length is i & j

(15) Longest common Substring

substring is continuous common string in given two string
i.e. if, string 1 is: a b c d e | Op: 2
string 2 is: a b f c e

then common substring will be: ab (not: abce (X))

Process:- same as LCS but there will be some changes in case 2:-

case 1:- when character is same of both string then LCS will be increase by 1. with previous length of substring.

case 2:- when character is not same of both string then [redacted] at that index length of longest common substring will be zero.

Note:- at last the length of longest common substring will be max of value in dp[i][j] (i.e iterate over all the possible length and return max)

Iterative implementation :-

```
int LCSubstr (string X, string Y, int m, int n)
    int DP[m+1][n+1];
    int result=0;
    for (int i=0; i<=m; i++)
        for (int j=0; j<=n; j++){
            if (i==0||j==0)
                DP[i][j]=0;
            else if (X[i-1]==Y[j-1]){
                DP[i][j]= DP[i-1][j-1]+1;
                result =max(result, DP[i][j]);
            }
            else
                DP[i][j]=0;
        }
    }
    return result;
}
```

// TC: O(m*n)
SC: O(m*n)

Space Optimized iterative implementation :-

in above code, we are only using the last row of the 2-D array. hence we can optimize the space by using a 2-D array of dimension 2*(min(n,m))

```
int LCSubstr (string X, string Y, int m, int n)
    int dp[2][n+1];
    int res=0;
    for (int i=1; i<=m; i++){
        for (int j=1; j<=n; j++){
            if (X[i-1]==Y[j-1]){
                dp[i%2][j]= dp[(i-1)%2][j-1]+1;
                if (dp[i%2][j]>res)
                    res=dp[i%2][j];
            }
            else dp[i%2][j]=0;
        }
    }
    return res;
}
```

Note:- here, n is longer or equal to m i.e string Y length is greater

// TC: O(m*n)
SC: O(min(m,n)).

(16) Printing longest common subsequence:

X: a b c d a f

Y: a r b c f

LCS DP table

		a	b	c	d	a	f
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

Print LCS

① if character at last index (m,n) is equal then add it into required subsequences. i.e ans = "f", and goes diagonally up.

② checks again for (m-1, n-1) as step 1. since the character is not same then don't add it to required subsequence and goes max (dp[i][j-1], dp[i-1][j]), and repeat step ① & ②

the required subsequence will be

Code:-

Note:- [redacted] using LCS problem code (page no. 146) create DP table. then. do Following.

```
int i=n, j=m;
String lcs="";
while (i>0 && j>0){
    if (X[i-1] == Y[j-1]){ lcs+=X[i-1]; i--; j--; }
    else{ if (dp[i][j-1] > dp[i-1][j]) j--;
          else i--;
    }
}
reverse (lcs.begin(), lcs.end());
```

supersequences :-

(17.) Shortest common ~~subsequences~~

- Given two string str1 and str2, the task is to find the length of the shortest string that has both str1 and str2 as subsequences.

ex:- str1 = "geek", str2 = "eke"

o/p: 5

explanation:- "geeke" has both string "geek" and "eke" as subsequences

process:-

we need to find a string that has both string as subsequences and is shortest such string. if both strings have all characters different, then result is sum of length of two given strings. if there are common characters, then we don't want them multiple times as the task is to minimize length. therefore we first find the longest common subsequence, take one occurrence of this subsequences and add extra characters

∴ Length of the shortest supersequence = (sum of lengths of given two strings) − (length of LCS of two given strings)

```
int SCS (string x, string y, int m, int n){
    return m+n - LCS(x,y,m,n);
}
```

Note:- LCS program is problem (9) page no. 145

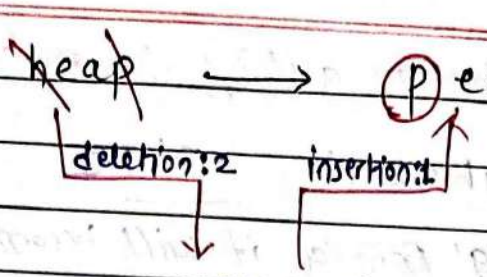
(18) minm no. of insertion & deletion to convert string a to string b.

Given two string 'str1' and 'str2' of size m and n respectively. the task is to remove/delete and insert the minm no. of character from/in str1 to transform it into str2.

ex:- str1 = "heap", str2 = "pea"

o/p:- minm deletion = 2 and minm insertion = 1

explanation: p & h deleted from str1 and again p is inserted in str2.



Process: Step1: take the LCS of both string.

Step2 :- delete the character from str1 which is not in LCS

i.e minm no. of deletion = m − length of LCS

Step3 :- Insert the character in str1 which is not in LCS in str2.

i.e minm no. of insertion = n − length of LCS

Code :-

```
void MinInsertDel (string x, string y, int n, int m){
    int lcs_len = LCS(x,y,m,n);
    cout<<"minm no. of deletion"<<(m-lcs_len)<<endl;
    cout<<"minm no. of insertions"<<(n-lcs_len)<<endl;
}
```

(19) Longest Palindromic Subsequence :-

- Given a string 's', Find the longest palindromic subsequence's length in 's'.
- ex:- (i) s="bbbab" | Explanation: "bbbb" is longest possible palindromic subsequence in this
  o/p= 4

(ii) s="cbbd" | (iii) GEEKSFORGEEKS
o/p= 2 | o/p=5
palindrome possible= EEKEE, EESEE, EEFEE,

Process:-

$$LPS(s) = LCS(s, \text{reverse}(s))$$

reverse the string and Find lcs of both. original & reverse.

(20) Minm no. of Deletion in a string to make it palindrome.

eg:- I/P : a = "agbcba" O/P: 1

when we will delete 'g' from 'a' it will become palindrome.

- we have given a string we have to delete minm character so that it become palindrome.

Process :-

$$\text{length of LPS} \propto \frac{1}{\text{no. of Deletion}}$$

- it means we have to make longest palindromic subsequence by deleting minm no. of character from given string

$$\therefore \text{min}^m \text{no. of deletion} = n - LPS(a)$$

(n = length of given string 'a')

and, $LPS = LCS(a, \text{reverse}(a))$

(21) Print Shortest common super sequence :-

- we have given two string and we have to print super sequence of that two given string.

ex:- I/P: a b c d a f
a c b c f

O/P: acbdcaf

Process :-

	Φ	a	b	c	d	a	f (Y)
Φ	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4
(X)							

acbcdaf

① when character are same include one of them and goes diagonally up. in LCS

② When character are not same then goes towards maxm(dp[i][j-1], dp[i-1][j]);

(i) If dp[i][j-1] is maxm [redacted] i.e we are at same row then include the character of vertical (ex:- abcdaf in this case) string, X[j] in SCS

(ii) if dp[i-1][j] is maxm i.e we are at same column then include the character X[i] in SCS.

③ if (i==0 or j==0) then (only one is zero)

(i) if i==0, then print X[j] & j-- until j become '0'

(ii) if j==0, then print X[i] & i-- until i become '0'.

Code:-

(length of X) (length of Y)
n m

```
string SCS (string x, string y, int n, int m){
    int dp[n+1][m+1]
    for (int i=0; i<=n; i++){
        for(int j=0; j<=m; j++){
            if (i==0 || j==0)
                dp[i][j]=0;
            if (i>=1 && j>=1){
                if (x[i-1]==y[j-1])
                    dp[i][j] = 1+dp[i-1][j-1];
                else
                    dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
            }
        }
    }
    int i=n, j=m;
    String scs="";
    while(i>0 && j>0){
        if(x[i-1]==y[j-1]){
            scs+=x[i-1]; i--; j--;
        }
        else if (dp[i][j-1] > dp[i-1][j]){
            scs+=y[j-1]; [redacted] j--;
        }
        else{
            scs+=x[i-1]; i--;
        }
    }
```

```
while (i>0){
    s[s++]=x[i-1]; i--;
}
while (j>0){
    s[s++]=y[j-1]; j--;
}
reverse(s.begin(), s.end());
return s;
}
```

(22) Longest repeating subsequence :-

- Given a string, find the length of the longest repeating subsequence such that the two subsequences don't have same string character at the same position, i.e any ith character in the two subsequences shouldn't have the same index in the original string.

Ex:- ① I/P: str = "AABEBCDD"
O/P: 3
The longest repeating subsequence is "ABD"

② I/P : str = "abc" O/P = 0, there is no repeating subsequence

③ I/P: str = "aab" O/P = 1, the two subsequence are 'a' (First) and 'a' (second).

Note:- 'b' can not be considered as part of subsequence as it could be at same index in both.

④ I/P: str = "aabb", O/P = 2, ab & ab

Process :- take 2 copy of same given string & find LCS of that two. and the index should not be equal of that character of LCS in the two string

a = "①A②B③EB④C⑤D⑥"
b = "①A②B③E④B⑤C⑥D⑦"

Change in code :-

Longest common subsequence code:	Longest repeating subsequence code
if (a[i-1]==b[j-1])	if (a[i-1]==b[j-1] && i!=j)
dp[i][j] = 1+dp[i-1][j-1];	dp[i][j] = 1+dp[i-1][j-1]
else	else
dp[i][j] = max(dp[i][j-1], dp[i-1][j]);	dp[i][j] = max(dp[i][j-1], dp[i-1][j]);

syb

(23) Sequence pattern matching

• we have given two string. we have to check whether one string is subsequence of another string

ex:- a = AXY   O/P: true
b = AOXCPY

Process: if(LCS(a,b) == min (a.length(), b.length())){
return true;
else
return false;   // TC: O(mn)

Note:- we can find this solution also by just a linear traversal. maintain two pointers at each string, move one if you find the character in another, at last if we traverse all the character of one string with condition that every string is present in another then o/p will be true   / TC: O(m+n)

(24) minimum number of insertion in a string to make it a palindrome.

ex:- s: "acbcbda"
O/P: 2   → acdbcbdca

Note:-	IP	Q	O/P	
this Qs.	one string	min#	int	matching 3/3
# of deletion	one string	min#	int	0/3 1/3 2/3 3/3 can be solved by # no. of deletion

Process:- (a) c (b) c (b) d (a)

• In minm no. of deletion problem we delete the extra character 'c' 'd'. and in minm no. of insertion we can insert this extra character again to make pair it means the minm no. of insertion will be equal to minm no. of deletion. i.e the both solution will be same.

i.e minm no. of deletion = minm no. of insertion = s.length − LPS

& LPS = LCS(s, reverse(s))

Partition DP - Problem

(25) Matrix chain multiplication :

mcm Based question
(i) mcm
(ii) Printing mcm
(iii) Evaluate Expr to True/Boolean Parenthesization
(iv) min/max value of an Expr
(v) Palindrome partitioning
(vi) Scramble string
(vii) Egg Dropping problem.

- Matrix chain multiplication is an optimization problem that to Find the most efficient way to multiply a given sequence of matrices. the problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplication involved.
- The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same. For example, For four matrices A, B, C, and D, we would have

$$((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))$$

- the order in which the product is parenthesized affects the no. of simple arithmetic operations needed to compute the product.

ex:- if 'A' is a (10x30) matrix | 'B' is (30x5) | 'C' is (5x60)

then, - A(BC) = (30x5x60) + (10x30x60) = 9000 + 18000 = 27000 operation

-(AB)C = (10x30x5) + (10x5x60) = 1500 + 3000 = 4500 operation

here it is clear that (AB)C is more efficient.

Problem:- we have given sequence of matrices dimension in an array arr[],

ex:- [40, 20, 30, 10, 30]

A = 40x20 | B = 20x30 | C = 30x10 | D = 10x30

the matrix 'i' = arr[i-1] x arr[i]

(i) i.e cost((AB)(CD)) = cost(AB) + cost(CD) + cost((AB)x(CD))

Date

Page No. 157.

(x) x (y)

(ii) Note:- cost of a single matrix i.e 'A' = 0

i.e for A(BC) = cost(A) + cost(BC) + cost(Ax(Y)) → Y = BxC

0

Subsequence

* Approach:-

- Take the sequence of matrices and separate it into two^
- Find minm cost to multiply each subsequence
- Add this cost together, and add cost of multiplying two matrix, result
- Do this for each possible position at which the sequence of matrices can be split, and take the minm over all of them

Forex:- For matrices ABCD, compute the cost by.

min ((A)(BCD), (AB)(CD), (ABC)(D))

↓ ↓ min((AB)C, A(BC))

and now, (BCD) cost = min (B(CD), (BC)D) Apply same concept. on subproblem

Note:- In this subproblem there will me more overlapping problem so we can use dynamic programming concept to optimize that.

* How to Seperate into two subsequence :-

ex:- arr: 0 → 10, i: 1 → 20 (A), 2 → 30 (B), 3 → 40 (C), j: 4 → 50 (D)

(1) k = (i → j-1)

k=1, f(i,k), f(k+1,j) → f(1,1), f(2,4)

i.e (A)(BCD)

k=2 f(1,2), f(3,4)

i.e (AB)(CD)

k=3 f(1,3), f(4,4)

i.e (ABC)(D)

(2) k = (i+1 → j)

f(i,k-1), f(k,j)

k=2, f(1,1), f(2,4)

i.e (A)(BCD)

k=3, f(1,2), f(3,4)

i.e (AB)(CD)

k=4, f(1,3), f(4,4)

i.e (ABC)(D)

* Base case :-

- there should be at least 3 element in array

i.e, i should be less than j

i.e if (i>=j) return 0;

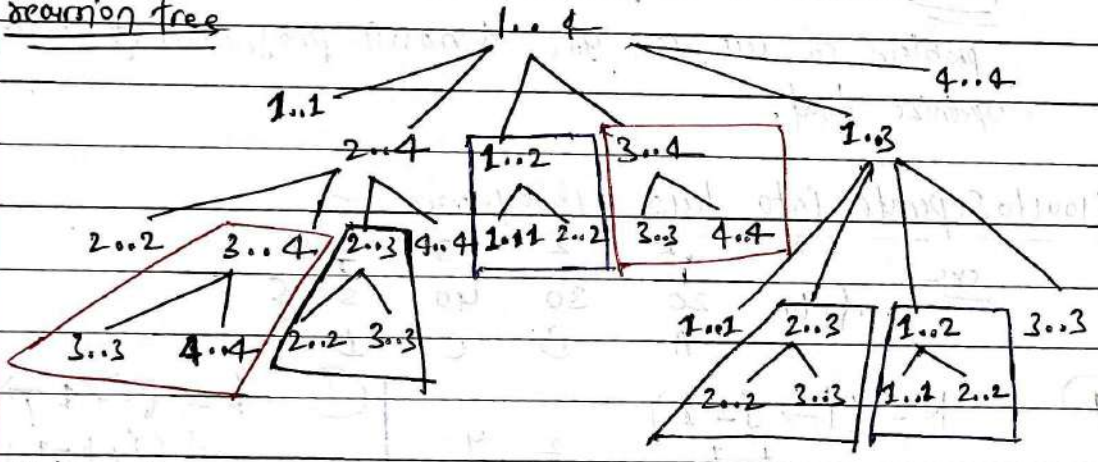
*Recursion code :— ↗ i=1 ↗ j=n-1 (starting)

```
int Solve (int arr[], int i, int j){
    if (i>=j) return 0;
    int ans = INT_MAX;
    for (int k=i; k<=j-1; k++){
        int temp_ans = Solve (arr, i,k) + Solve (arr, k+1,j) +
                        arr[i-1] * arr[k] * arr[j];
        ans = min (ans, temp_ans);
    }
    return ans;
}
```

TC: exponential

* Dynamic programming code:— (Memoization)

recursion tree



Note:- there are overlapping sub-problem so we can store the answer as dp and used later whenever need.

```
int t[1000][1000];
int solve (int arr[], int i, int j){
    if (i>=j){
        t[i][j]=0;
        return 0;
    }
    if (t[i][j] != -1)
        return t[i][j];
    int ans = INT_MAX;
```

```
for(int k=1; k<=j-1;k++){
    int temp_ans= solve(arr, i, k) + solve(arr, k+1, j)+
                        arr[i-1]*arr[k]*arr[j];
    ans= min(ans, temp_ans);
}
return t[i][j]= ans;
}
int helper(int arr[], int n){
    memset(t, -1, sizeof(t));
    return solve(arr, 1, n-1);
}
```

TC: $O(N^3)$

SC: $O(N^2)$ Ignoring recursion stack space

* Tabulation method (Iterative Approach):-

- for each $2<=k<=n$, the minm cost of all subsequences of length k, using the cost of smaller subsequences already computed, has same asymptotic runtime and no recursion.

```
int solve(int arr[], int n){
    int t[n+1][n+1];
    for(int i=1; i<=n; i++){
        t[i][i]=0;
    }
    for(int len=2; len<=n; len++){
        for(int i=1; i<=n-len+1; i++){
            int j= i+len-1;
            t[i][j]=INT_MAX;
            for(int k=i; j<n && k<=j-1; k++){
                int cost= t[i][k]+t[k+1][j]+arr[i-1]*arr[k]*
                                arr[j];
                if(cost<t[i][j])
                    t[i][j]= cost;
            }
        }
    }
    return t[1][n-1];
}
```

(26.) **Palindrome Partitioning**

Given a string, determine the fewest cuts/partitioning of string so that every substring of the partition become a palindrome

ex:- S = "ababbbabbababa"

partition = 3

"a|babbbab|b|ababa"

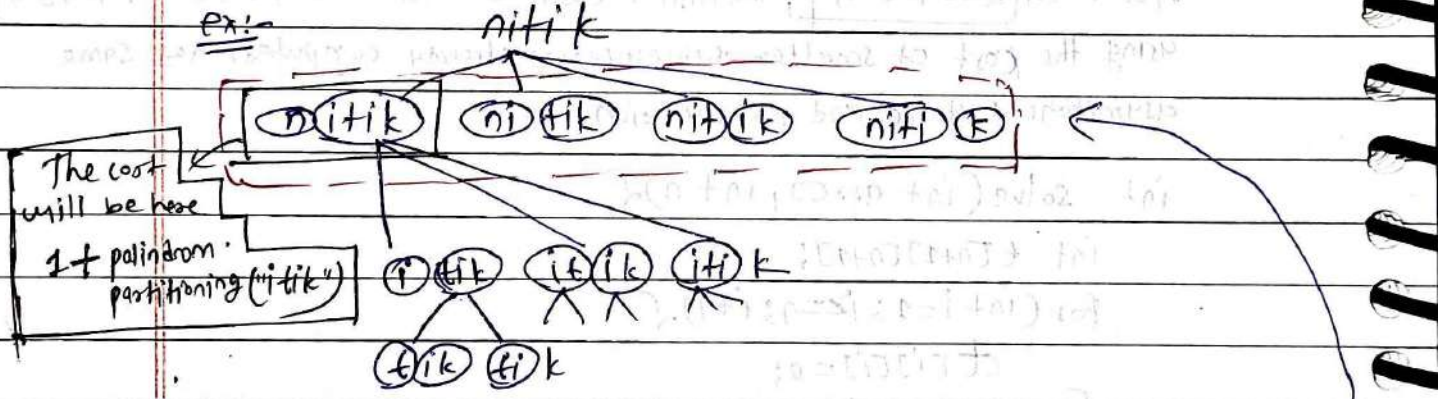
Note:- . if given string is already palindrome then partition = 0.

. if all the character are different then partition = n-1.

n = length of string

Approach:- like the matrix chain multiplication problem, use try making cuts at all possible places, recursively calculate the cost for each cut and return the minm value.

ex:-



Note:- here, palindrome partioning for nitik is minm of all palindrome partitionings or

* find i and j
* find BC
* find k loop

⇒ change in mcm (matrix chain multiplication

* find i and j

n i t i k

i j (i=0, j=n-1)

. we no need to start i from 1 as there is no required of (i-1)

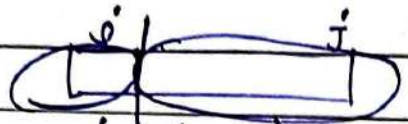
* Base case:-

i == j, size = 1 → return 0;

i > j, size = 0 → if (empty string or one character then it is palindrome no need of partition hence partition is zero.

* find 'k' loop

(1)

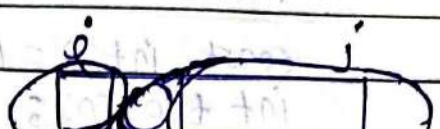


$f_1(i \text{ to } k), f_2(k+1 \text{ to } j)$

- if $k=j$ it means $(k+1 \text{ to } j)$ not possible hence last value of k will be $j-1$.

hence k loop is
$k(i \rightarrow j-1)$

(2)



$f_1(i \text{ to } k-1)$ $f_2(k \text{ to } j)$

- here last value of $k=j$ is possible.
because $f_1(i \text{ to } j-1)$ & $f_2(j \text{ to } j)$

hence k loop is
$k(i+1 \rightarrow j)$

* Recursive Code

```
bool isPalindrome(string x, int i, int j){
    while(i<=j){
        if(x[i]!=x[j]) return false;
        i++, j--;
    }
    return true;
}

int solve(string x, int i, int j){
    if(i>=j || isPalindrome(x,i,j)) return 0;
    int ans=INT_MAX;
    for(int k=i; k<j; k++){
        int temp_ans= solve(x,i,k)+solve(x,k+1,j)+1;
        ans=min(ans, temp_ans);
    }
    return ans;
}
```

TC: exponential

* Memoization (DP):-

```
const int D = 1001;
int t[D][D];

bool isPalindrome (string x, int i, int j){
    while (i<=j){
        if(x[i] != x[j]) return false;
        i++; j--;
    }
    return true;
}

int solve (string x, int i, int j){
    if (i>=j || isPalindrome (x,i,j)){
        return t[i][j]=0;
    }
    [illegible]
    if(t[i][j] != -1) return t[i][j];
    int ans = INT_MAX;
    for(int k=i; k<j; k++){
        int temp_ans = solve(x,i,k) + solve (x, k+1, j) + 1;
        ans = min (ans, temp_ans);
    }
    return t[i][j] = ans;
}
//TC : O(n^3)
```

* memoized optimized :-

```
int solve (string x, int i, int j){
    if (t[i][j] != -1) return t[i][j];
    if [i>=j || isPalindrome (x, i, j)){
        t[i][j] = 0;
        return 0;
    }
    int ans = INT_MAX;
    for (int k=i; k<j; k++){
        int left, right;
```

```
if(t[i][k] == -1)
        left = solve(x, i, k);
else
        left = t[i][k];
if(t[k+1][j] == -1)
        right = solve(x, k+1, j);
else
        right = t[k+1][j];
int temp_ans = left + right + 1;
ans = min(ans, temp_ans);
}
return t[i][j] = ans;
}
```

// TC: O(n³)

(27) Evaluate Expression to True Boolean Parenthesization:-

Given a boolean expression with 'T' & 'F' symbol and operator &, |, ^ count the no. of ways we can parenthesize the expression so that the value of expression evaluates to true.

ex: ① expression = "T^F&T"
o/p = 2 "(T^F)&T)" and "(T^(F&T))"

② expression = "T|T&F^T"
o/p = 4 "((T|T)&(F^T))", "(T|(T&(F^T)))", "(((T|T)&F)^T)" and "(T|((T&F)^T))".

Approach:-

In mcm we put bracket for min cost & in this also we put the bracket for true evaluation. We need to break bracket on operator. expn operator expn

4-step:-
① find i & j
② find base condn
③ Find k loop
④ temp ans & find

Find i, j T|F&T∧F

(1) i=0 , j= str.length()-1

(2) BC

(T | F & T) ∧ (F)

i to k-1 — k — k+1 to j

if (i > j) return false
if (i==j){ → it means this is one character
    if (is True) return True.
    else return false
}

Note:- k will be always at operator (|, &, ∧). and i and j will be at T or F.

(3) Find k loop:-
k will be always after 2 step bcz operator is b/w two 'T' or 'F'

For (int k=i+1 ; k<=j-1; k=k+2)

(4) Temp ans:-
- It will return left true, left false, right true and right false

```
For (int k=i+1; k<=j-1; k=k+2)
    int lT = (s, i, k-1, T)
    int lF = (s, i, k-1, F)
    int RT = (s, k+1, j, T)
    int RF = (s, k+1, j, F)
```
temp ans.

Now, we have three operator (&, |, ∧)

```
∴ if (s[k] == '&')
      if (isTrue == True)  ans = ans + lT*RT
      else  ans = ans + lF*RT + lF*RF + lF*RF
  else if (s[k] == '|')
      if (isTrue == True)  ans = ans + lT*RT + lT*RF + lF*RT
      else  ans = ans + lF*RF
  else
      if (isTrue == True)  ans = ans + lF*RT + lT*RF
}
return ans;
```

Recursive Code:-

```
int solve(string x, int i, int j, bool isTruee){
    if(i>=j){
        if(isTrue) return x[i]=='T';
        else       return x[i]=='F';
    }
    int ans=0;
    for(int k=i+1; k<j; k+=2){
        int l_T=solve(x,i,k-1,true);
        int l_F=solve(x,i,k-1,false);
        int r_T=solve(x,k+1,j,true);
        int r_F=solve(x,k+1,j,false);
        if(x[k]=='|'){
            if(isTrue==true)
                ans+= l_T*r_T+l_T*r_F+l_F*r_T;
            else
                ans+=l_F*r_F;
        }
        else if(x[k]=='&'){
            if(isTrue==true) ans+=l_T*r_T;
            else ans+=l_T*r_F+l_F*r_T+l_F*r_F;
        }
        else if(x[k]=='^'){
            if(isTrue==true) ans+=l_T*r_F+l_F*r_T;
            else ans+=l_T*r_T+l_F*r_F;
        }
    }
    return ans;
}
```

(Memoization) Approach

- since there are three variable which is changing (i, j and isTrue) so we have to use 3d array to using memoization concept. i and j will be of string length and isTrue will be (T & F)

```
#include <bits/stdc++.h>
using namespace std;
const int D = 1001;
int dp[2][D][D];
int solve(string x, int i, int j, bool isTrue){
    if (i>=j){
        if(isTrue) dp[1][i][j] = x[i]=='T';
        else       dp[0][i][j] = x[i]=='F';
        return dp[isTrue][i][j];
    }
    if (dp[isTrue][i][j] != -1)
        return dp[isTrue][i][j];
    int ans = 0;
    for(int k=i+1; k<j; k+=2){
        int l_T = solve(x, i, k-1, true);
        int l_F = solve(x, i, k-1, false);
        int r_T = solve(x, k+1, j, true);
        int r_F = solve(x, k+1, j, false);
        if (x[k]=='|'){
            if(isTrue==true) ans += l_T*r_T + l_T*r_F + l_F*r_T;
            else ans += l_F*r_F;
        }else if(x[k]=='&'){
            if(isTrue==true) ans += l_T*r_T;
            else ans += l_T*r_F + l_F*r_T + l_F*r_F;
        }else if(x[k]=='^'){
            if(isTrue==true) ans += l_T*r_F + l_F*r_T;
            else ans += l_T*r_T + l_F*r_F;
        }
    }
    dp[isTrue][i][j] = ans;
    return ans;
}
int main(){
    string x; cin>>x;
    memset(dp[0], -1, sizeof(dp[0])); memset(dp[1], -1, sizeof(dp[1]));
    cout<<solve(x, 0, x.length()-1, true)<<endl;
```

we can optimize this by first checking if corresponding function returning result are present or not in dp table i.e if dp[isTrue][i][j] != -1 then call the function

(*) Memoization Approach using map instead of 3D array.

```
unordered_map<string, int> ump;
int solve (string x, int i, int j, bool isTrue){
    string key = to_string(i) + " " + to_string(j) + " " + (isTrue?"T":"F");
    if (ump.find(key) != ump.end()) return ump[key];
    if (i>=j){
        if (isTrue) ump[key] = x[i]=='T';
        else ump[key] = x[i]=='F';
        return ump[key];
    }
    int ans=0;
    for (int k=i+1; k<j; k+=2){
        int l_T = solve(x, i, k-1, true);
        int l_F = solve(x, i, k-1, false);
        int r_T = solve(x, k+1, j, true);
        int r_F = solve(x, k+1, j, false);
        if (x[k]=='|'){
            if (isTrue==true) ans+= l_T*r_T + l_T*r_F + l_F*r_T;
            else ans+= l_F*r_F;
        }
        else if (x[k]=='&'){
            if (isTrue==true) ans+= l_T*r_T;
            else ans+= l_T*r_F + l_F*r_T + l_F*r_F;
        }
        else if (x[k]=='^'){
            if (isTrue==true) ans+= l_T*r_F + l_F*r_T;
            else ans+= l_T*r_T + l_F*r_F;
        }
    }
    return ump[key] = ans;
}
signed main(){
    string x; cin>>x;
    ump.clear();
    cout<<solve(x, 0, x.length()-1, true)<<endl;
}
```

Date
Page No. 168.

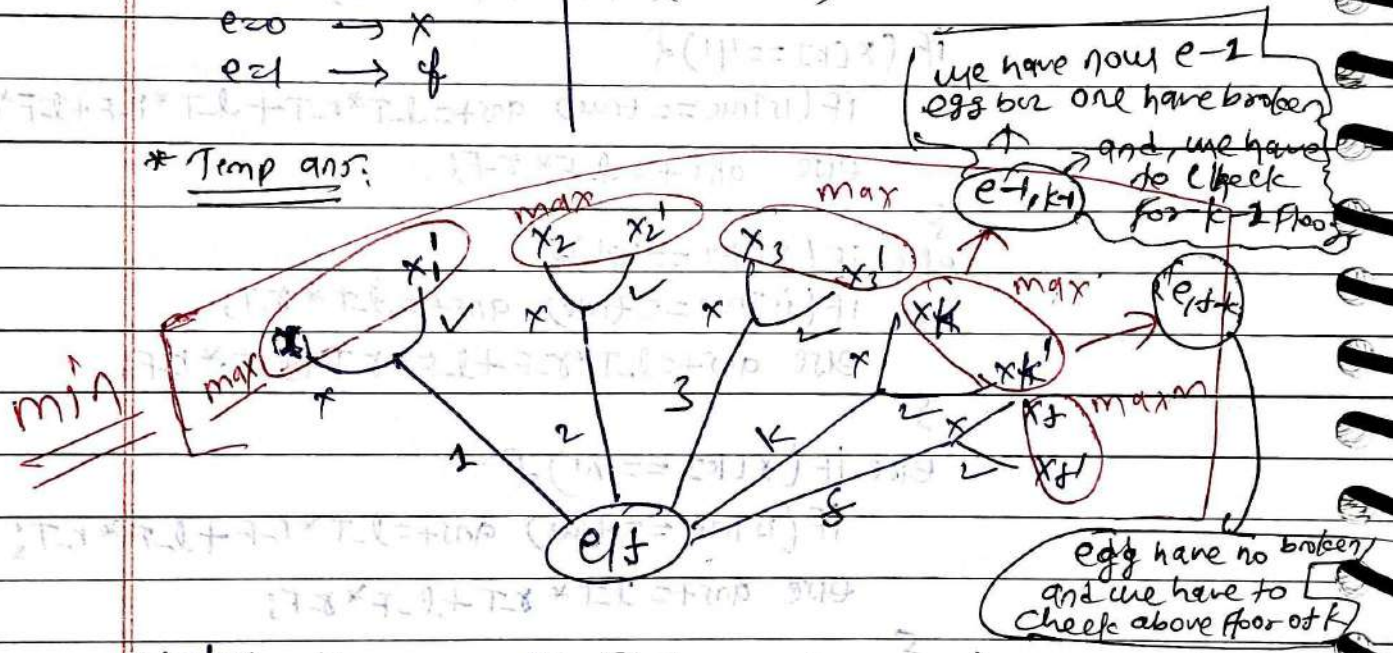
(28.) Egg Dropping problem :-

- you are given e identical eggs and you have access to a building with f floor labeled from 1 to f.
- you know that there exists a floor f' where $0 <= f' <= f$ such that any egg dropped at a floor higher than f' will break and any egg dropped at or below floor "f'" will not break.
- Each move, you may take an unbroken egg and drop it from any floor 'x' (where $1 <= x <= f$). If the egg breaks, you can no longer use it. However if the egg does not break, you may reuse it in future moves.
- Return the minimum number of moves that you need to determine with certainty what the value of f' is.

* Base condition
  - $f=0 \rightarrow 0$
  - $f=1 \rightarrow 1$
  - $e=0 \rightarrow \infty$
  - $e=1 \rightarrow f$

* k, loop
  - $k=(1 \rightarrow f)$

* Temp ans:



Note:- First we will find worst case in every partition and then for all worst case we will find min of them.

Recursion code:-

we have given total eggs and total floors:.

```
int Solve(int eggs, int floors){
    if(eggs==1) return floors;
    if(floors==0||floors==1) return floors;
    int mn=INT_MAX;
    for(int k=1;k<=floors;k++){
        int temp_ans=1+max(Solve(eggs-1,k-1),
                           solve(eggs,floors-k));
        mn=min(mn,temp_ans);
    }
    return mn;
}
```

*memoization code:-

```
const int D=101;
int t[D][D];
int Solve(int eggs, int floors){
    if(t[eggs][floors]!=-1) return t[eggs][floors];
    if(eggs==1||floors==0||floors==1){
        t[eggs][floors]=floors;
        return floors;
    }
    int mn=INT_MAX;
    for(int k=1;k<=floors;k++){
        int temp_ans=1+max(Solve(eggs-1,k-1),
                           Solve(eggs,floors-k));
        mn=min(mn,temp_ans);
    }
    return t[eggs][floors]=mn;
}
int main(){
    int eggs,floors; cin>>eggs>>floors;
    memset(t,-1,sizeof(t));
    cout<<Solve(eggs,floors)<<endl;
}
```