Dynamic Programming

Memoization {Bottom - Up} → Tabulation {Top - Down} → Space Optimization

Memoization : We store the answers of subproblems, as we tend to overlap them

Tabulation : Memoization without recursion {recursion stack is excluded ⇒ space reduced}

Space Optimization : Tabulation without storing {We rather use constant space like prev1 & prev2 ⇒ space much more reduced}. This is not always feasible considering possible stuffs we can do.

- Count the total number of ways
- There are multiple ways of doing it, given me which one leads to min / max

1. Try and visualize the problem in terms of indices
2. Do all possible stuff on each index
3. Finally return what is asked from all stuffs

Greedy : Focuses only on the next best option and not good in long run

```
#include <bits/stdc++.h>

// int f(int n, vector<int> &heights, vector<int> &dp) {

//     if(n == 0) return 0;
//     if(dp[n] != -1) return dp[n];

//     int p1 = f(n-1, heights, dp) + abs(heights[n] - heights[n-1]);

//     int p2 = INT_MAX;
//     if(n > 1) p2 = f(n-2, heights, dp) + abs(heights[n] - heights[n-2]);

//     return dp[n] = min(p1, p2);
// }

int frogJump(int n, vector<int> &heights)
{

    // return f(n-1, heights, dp);
    // vector<int> dp(n, -1);
    int prev1 = 0;
    int prev2 = 0;
    for(int i = 1;i < n;i++) {
```

```
        int p1 = prev1 + abs(heights[i] - heights[i-1]);
        int p2 = INT_MAX;
        if(i > 1) p2 = prev2 + abs(heights[i] - heights[i-2]);

        int curr = min(p1, p2);
        prev2 = prev1;
        prev1 = curr;
    }

    return prev1;
}
```

- From recursion → memoization

1. declare dp array
2. pass the dp array in recursion
3. first store and then return
4. add an additional base case to check if the subproblem has already been solved

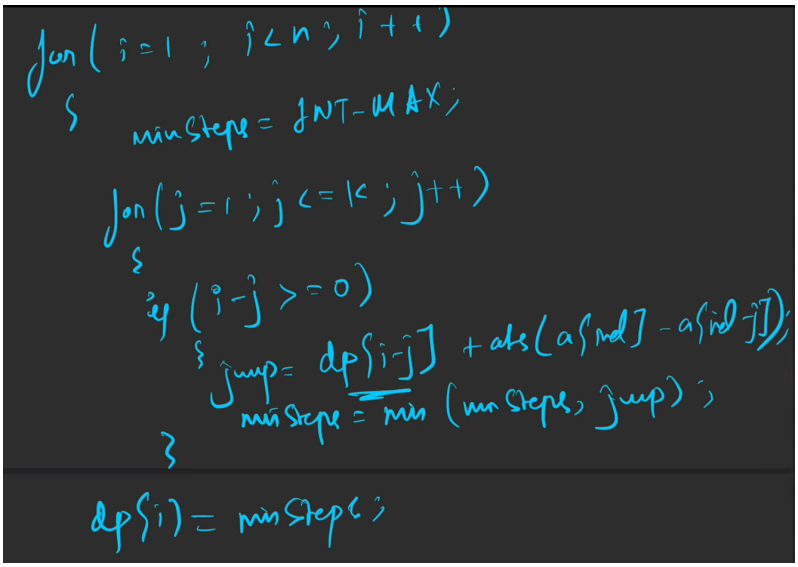
- memoization → tabulation

1. keep the dp array
2. convert the base cases into declarations
3. run a for loop omitting the declared base cases {int i = 1 type} such that all states can be expressed in it

- space optimization :

  declare number of variable to possible stuff we can do, then keep updating them while iterating. answer will prev.

If we can do K stuff -



- here j represents number of steps, i - j to make sure we don't go less than 0th step and keep updating min from possible steps and return the smallest.

In case of dealing with subsequence problems do it with pick/non-pick

```
int pick = nums[n];
if(n-2 >= 0) pick +=  f(n-2, nums, dp);
int non_pick = f(n-1, nums, dp);

return dp[n] = max(pick, non_pick);
```

in pick we are adding f(n-2) rather than f(n-1) because we are not allowed to take adjacent indices.

- In case the houses were circular, meaning the first and last houses are neighbors then we apply the same logic twice once with 1 excluding last, then including last and excluding. Finally return the max of both.

2D-DP 🙂

```
#include<bits/stdc++.h>

// int f(int n, vector<vector<int>> &points, int last,
// vector<vector<int>> &dp) {
//     // base case
//     if(n == 0) {
//         int maxi = INT_MIN;
//         for(int i = 0;i < 3;i++) {
//             if(i != last) maxi = max(maxi, points[0][i]);
//         }
//         return maxi;
//     }

//     if(dp[n][last] != -1) return dp[n][last];

//     int maxi = INT_MIN;
//     for(int i = 0;i < 3;i++) {
//         if(i != last) {
//             maxi = max(maxi, points[n][i] + f(n-1, points, i, dp));
//         }
//     }

//     return dp[n][last] = maxi;
// }

int ninjaTraining(int n, vector<vector<int>> &points)
{
```

```
    // return f(n-1, points, 3, dp);

    // vector<vector<int>> dp(n, vector<int>(4, INT_MIN));
    // dp[0][0] = max(points[0][1], points[0][2]);
    // dp[0][1] = max(points[0][0], points[0][2]);
    // dp[0][2] = max(points[0][0], points[0][1]);
    // dp[0][3] = max({points[0][0], points[0][1], points[0][2]});

    vector<int> prev_day(4, INT_MIN);
    prev_day[0] = max(points[0][1], points[0][2]);
    prev_day[1] = max(points[0][0], points[0][2]);
    prev_day[2] = max(points[0][0], points[0][1]);
    prev_day[3] = max({points[0][0], points[0][1], points[0][2]});


    for(int day = 1; day < n; day++) {
        vector<int> temp(4, INT_MIN);
        for(int prev_task = 0;prev_task <= 3;prev_task++) {
            for(int curr_task = 0;j < 3;j++) {
                if(curr_task != prev_task) { // no repetition
                    // dp[day][prev_task] = max(dp[day][prev_task],
                                // points[day][curr_task] + dp[day-1][curr_task])
                    temp[prev_task] = max(temp[prev_task],
                     points[day][curr_task] + prev[curr_task]);
                }
            }
        }
        prev_day = temp;
    }

    // return dp[n-1][3];
    return prev_day[3];
}
```

1. Here every day is considered an index
2. all curr_task we can do on a day depends on what we did prev_task
3. make sure curr_task ≠ prev_task and calc max based on that
4. one important edge case is prev_task can be 3 {where we can choose any task from 0, 1, 2}

In case of 2D dp during space optimization we take a row vector as prev, and a temp row vector to deal with current stuff and then late we make prev = temp.

```
5// top-left to bottom-right rec func for unique paths
int f(int m, int n, vector<vector<int>> &dp) {
    if(m == 0 && n ==0) return 1;
    // when we cross boundaries
```

```
    if(m < 0 || n < 0) return 0;

    // already solved subproblem
    if(dp[m][n] != -1) return dp[m][n];
    // recursion
    int up = f(m-1, n, dp);
    int left = f(m, n-1, dp);

    return dp[m][n] = up + left;
}
```

```
int minSumPath(vector<vector<int>> &grid) {
    // top-left to bottom-right traversal
    int m = grid.size();
    int n = grid[0].size();
    vector<int> prev(n, 0);
    for (int i = 0; i < m; i++) {
        vector<int> temp(n, 0);
        for (int j = 0; j < n; j++) {
            if (i == 0 && j == 0) {
                temp[j] = grid[0][0];
            } else {
                    // Take care of boundary case {IMP}
                int up = (i > 0) ? prev[j] + grid[i][j] : INT_MAX;
                int left = (j > 0) ? temp[j - 1] + grid[i][j] : INT_MAX;
                temp[j] = min(up, left);
            }
        }
        prev = temp;
    }
    return prev[n - 1];
}
```

- In case there are multiple starting point // ending points depending upon recursion or iteration on must take care of returning amongst all answers min or max as shown

```
#include <bits/stdc++.h>

// int f(int row, int col, vector<vector<int>>& triangle, int n,
// vector<vector<int>> &dp) {
//  if(row == 0 && col == 0) return triangle[0][0];
//  if(dp[row][col] != -1) return dp[row][col];

//  int up = triangle[row][col];
//  if(col != row) up += f(row-1, col, triangle, n, dp);
```

```
//  else up = INT_MAX;

//  int dia_up = triangle[row][col];
//  if(col > 0) dia_up += f(row-1, col-1, triangle, n, dp);
//  else dia_up = INT_MAX;

//  return dp[row][col] = min(up, dia_up);
// }

int minimumPathSum(vector<vector<int>>& triangle, int n){
    // int mini = INT_MAX;
    // vector<vector<int>> dp(n, vector<int>(n, -1));
    // for(int col = n-1;col >= 0;col--) {
    //  mini = min(mini, f(n-1, col, triangle, n, dp));
    // }
    // return mini;

    vector<int> prev(n, 0);
    for(int row = 0;row < n;row++) {
        vector<int> curr(n, 0);
        for(int col = 0;col <= row;col++) {
            if(row == 0 && col == 0) curr[col] = triangle[row][col];
            else {
                int up = triangle[row][col];
                if(col != row) up += prev[col];
                else up = INT_MAX;

                int dia_up = triangle[row][col];
                if(col > 0) dia_up += prev[col-1];
                else dia_up = INT_MAX;

                curr[col] = min(up, dia_up);
            }
        }
        prev = curr;
    }
    return *min_element(prev.begin(), prev.end());
}
```

- In case we have to move in multiple ways

```
int minFallingPathSum(vector<vector<int>> &vec, int n) {

    vector<int> prev(n, 0);

    for(int row = 0;row < n;row++) {
        vector<int> curr(n, 0);
```

```
        for(int col = 0;col < n;col++) {
            if(row == 0) curr[col] = vec[row][col];
            else {
                int up_left = INT_MAX, up_right = INT_MAX;

                if (col > 0) up_left = vec[row][col] + prev[col - 1];
                int up = vec[row][col] + prev[col];
                if (col < n - 1) up_right = vec[row][col]
                                                                        + prev[co

                curr[col] = min({up_left, up, up_right});
            }
        }
        prev = curr;
    }

    return *min_element(prev.begin(), prev.end());
}
```

- SIMULTANEOUS DP on two starting points :

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

// int f(int i, int j1, int j2, int r, int c, vector<vector<int>> &grid,
// vector<vector<vector<int>>> &dp) {
//     if (j1 < 0 || j1 >= c || j2 < 0 || j2 >= c) return INT_MIN;

//     if (i == r-1) {
//         if (j1 == j2) return grid[i][j1];
//         else return grid[i][j1] + grid[i][j2];
//     }

//     if (dp[i][j1][j2] != -1) return dp[i][j1][j2];

//     int maxi = INT_MIN;
//     for (int dj1 = -1; dj1 <= 1; dj1++) {
//         for (int dj2 = -1; dj2 <= 1; dj2++) {
//             int new_j1 = j1 + dj1;
//             int new_j2 = j2 + dj2;
//             int value;
//             if (new_j1 == new_j2) value = grid[i][j1];
//             else value = grid[i][j1] + grid[i][j2];
//             value += f(i+1, new_j1, new_j2, r, c, grid, dp);
//             maxi = max(maxi, value);
//         }
//     }
```

```
//     return dp[i][j1][j2] = maxi;
// }

int maximumChocolates(int n, int c, vector<vector<int>> &grid) {
    // vector<vector<vector<int>>>
      //              dp(n, vector<vector<int>>(c, vector<int>(c, -1)));
    vector<vector<int>> prev(c, vector<int>(c, 0));

    for (int j1 = 0; j1 < c; j1++) {
        for (int j2 = 0; j2 < c; j2++) {
            if (j1 == j2)
                prev[j1][j2] = grid[n-1][j1];
            else
                prev[j1][j2] = grid[n-1][j1] + grid[n-1][j2];
        }
    }

    for (int i = n-2; i >= 0; i--) {
        vector<vector<int>> curr(c, vector<int>(c, 0));
        for (int j1 = 0; j1 < c; j1++) {
            for (int j2 = 0; j2 < c; j2++) {
                int maxi = INT_MIN;
                for (int dj1 = -1; dj1 <= 1; dj1++) {
                    for (int dj2 = -1; dj2 <= 1; dj2++) {
                        int new_j1 = j1 + dj1;
                        int new_j2 = j2 + dj2;
                        if (new_j1 >= 0 && new_j1 < c &&
                                            new_j2 >= 0 && new_j2 < c) {
                            int value;
                            if (new_j1 == new_j2)
                                value = grid[i][j1];
                            else
                                value = grid[i][j1] + grid[i][j2];
                            value += prev[new_j1][new_j2];
                            maxi = max(maxi, value);
                        }
                    }
                }
                curr[j1][j2] = maxi;
            }
        }
        prev = curr;
    }

    return prev[0][c-1];
}
```

- DP on subsequences/subarrays : Generally pick and not-pick wala concept lagta he

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

bool f(int ind, int target, vector<int> &arr, vector<vector<int>> &dp) {
    if (target == 0) return true;
    if (ind == 0) return (arr[0] == target);

    if (dp[ind][target] != -1) return dp[ind][target];

    bool notpick = f(ind - 1, target, arr, dp);
    bool pick = false;
    if (arr[ind] <= target) pick = f(ind - 1, target - arr[ind], arr, dp);

    return dp[ind][target] = notpick || pick;
}

bool subsetSumToK(int n, int k, vector<int> &arr) {
    // vector<vector<bool>> dp(n, vector<bool>(k + 1, false));
    // return f(n - 1, k, arr, dp);

    vector<bool> prev(k+1, false);
    prev[0] = true;
    if(arr[0] <= k) prev[arr[0]] = true;

    // for(int i = 0;i < n;i++) dp[i][0] = true;
    // dp[0][arr[0]] = true;

    for(int ind = 1; ind < n; ind++) {
        vector<bool> curr(k+1, false);
        curr[0] = true;
        for(int target = 0; target <= k; target++) {
            bool notpick = prev[target];
            bool pick = false;
            if (arr[ind] <= target) pick = prev[target - arr[ind]];

            curr[target] = notpick || pick;
        }
        prev = curr;
    }
    return prev[k];
}
```

- Here in recursion, there will be some target values which we will not encounter but in case of iterative solution we calculate for all possible target values {HENCE IN WORST CASE they will run in same time, else recursive is faster with an extra auxiliary space}

- Also here there were two base cases and hence two declarations were made in tabulation

1. **Partition**: Divides a set into disjoint subsets. {1, 2, 3}, a possible partition is {{1, 2}, {3}}
2. **Subset**: Any selection of elements from a set. {1, 2, 3}, possible subsets include {}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, and {1, 2, 3}
3. **Subarray**: A contiguous segment of an array. [1, 2, 3], possible subarrays include [1], [2], [3], [1, 2], [2, 3], and [1, 2, 3]
4. **Subsequence**: A sequence derived by deleting some or no elements, maintaining order. [1, 2, 3], possible subsequences include [], [1], [2], [3], [1, 2], [1, 3], [2, 3], and [1, 2, 3]

- so subset and and subsequence are same
- subarray cannot be empty

- One important property of partition is that if we know the sum of one set, then other can be calculated as they are disjoint
- say we have [0] then number of subsets whose sum is zero are two : [] and [0]
- And the rest will take care on its on in pick and non-pick case

```
if(!arr[0]) prev[arr[0]] = 2;
else if (arr[0] <= k) prev[arr[0]] = 1;
```

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

// Function to compute the minimum number of elements needed to sum up to x
int f(int index, int x, vector<int> &num, vector<vector<int>> &dp) {
    if (x == 0) return 0; // No need to further traversal
    if (index == 0) {
        // This is the last number
        return (x % num[0] == 0) ? x / num[0] : INT_MAX;
    }

    if (dp[index][x] != -1) return dp[index][x];

    int not_pick = f(index - 1, x, num, dp);

    // Pick the current element (if possible)
    // AND STAY AT THE SAME INDEX SO WE CAN REPICK
    int pick = INT_MAX;
    if (num[index] <= x) {
        int res = f(index, x - num[index], num, dp);
        if (res != INT_MAX) {
        // AS WE MAY RETURN INT_MAX ADDING 1 WILL CAUSE OVERFLOW !
```

```
            pick = 1 + res;
        }
    }

    return dp[index][x] = min(not_pick, pick);
}

int minimumElements(vector<int> &num, int x) {
    int n = num.size();

    vector<int> prev(x + 1, INT_MAX); REMEMBER x+1 BECAUSE 0<=target<=x
    prev[0] = 0;

    for (int t = 0; t <= x; t++) {
        if (t % num[0] == 0) prev[t] = t / num[0];
    }

    for (int i = 1; i < n; i++) {
        vector<int> curr(x + 1, INT_MAX);
        curr[0] = 0; BASE CASE INSIDE LOOP ASWELL

        for (int t = 0; t <= x; t++) {
            int not_pick = prev[t];
            int pick = INT_MAX;

            if (num[i] <= t) {
                int res = curr[t - num[i]];
                if (res != INT_MAX) {
                    pick = 1 + res;
                }
            }

            curr[t] = min(not_pick, pick);
        }

        prev = curr;
    }

    return (prev[x] == INT_MAX) ? -1 : prev[x];
}
```

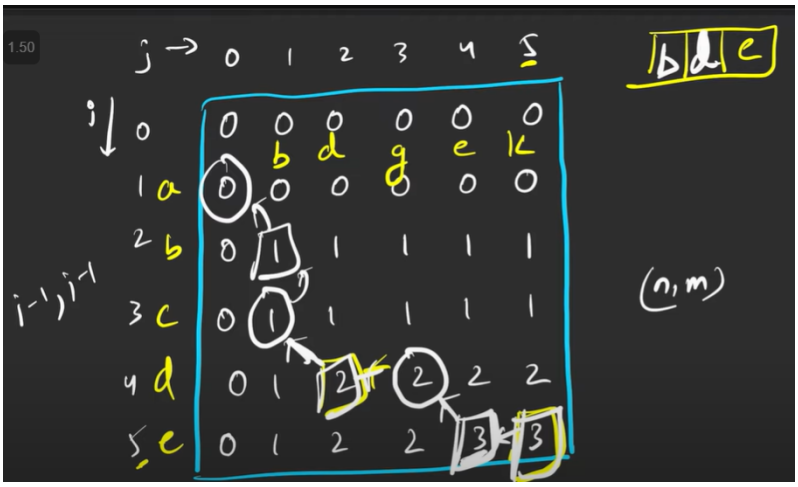
DP on STRINGS

```
class Solution {
public:
    int f(int n1, int n2, string &text1, string &text2,
                                                                    vector<vector<in

        if (n1 == 0 || n2 == 0) return 0;
```

```
        if(dp[n1][n2] != -1) return dp[n1][n2];

        if(text1[n1-1] == text2[n2-1])
        // IF MATCHING SHIFT BOTH BY 1
            return dp[n1][n2] = 1 + f(n1-1, n2-1, text1, text2, dp);
        else
        // ELSE TRY BOTH ONE AT A TIME
            return dp[n1][n2] = max(f(n1-1, n2, text1, text2, dp), f(n1, n2-1, text1,
    }

    int longestCommonSubsequence(string text1, string text2) {
        int n1 = text1.length();
        int n2 = text2.length();
        // vector<vector<int>> dp(n1+1, vector<int>(n2+1, 0));
        return f(n1, n2, text1, text2, dp)
        // WE ARE PASSING n1 & n2 TO MAKE BASE CASE IN TABULATION EASY
        vector<int> prev(n2+1, 0);
        // WHEN WE TRY SPACE OPTIMISING MAKE SURE THE SIZE OF PREV AND
        // CURR ARE SET TO LIMIT OF SECOND VARIABLE

        for(int i = 1; i <= n1; i++) {
            vector<int> curr(n2+1, 0);
            curr[1] = 0;
            for(int j = 1; j <= n2; j++) {
                if(text1[i-1] == text2[j-1]) {
                    curr[j] = 1 + prev[j-1];
                }
                else {
                    curr[j] = max(prev[j], curr[j-1]);
                }
            }
            prev = curr;
        }
        return prev[n2];
    }
};
```

How do we get that longest common subsequence ?

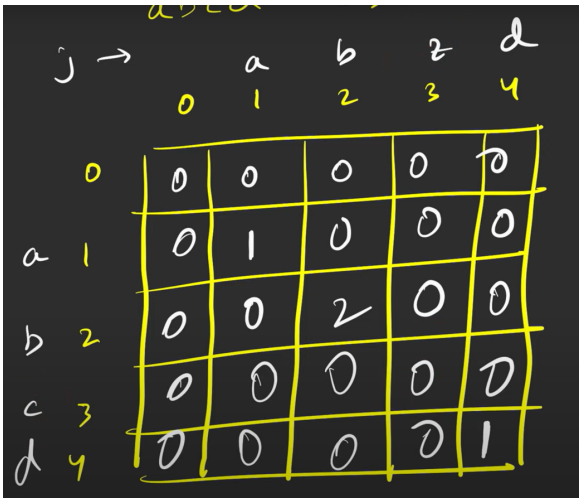
- Use our dp array :



This shows how tabulation works :

- Either a diagonal jump when element mathces
- or we move right or down {which ever is max}
- Just backtrack and see where we have jumped diagonally and add it in string

Now say we want longest common substring {characters have to be consecutive}



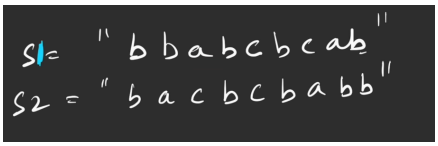
dp[n1][n2] = max(f(n1-1, n2, text1, text2, dp), f(n1, n2-1, text1, text2, dp); is changed to

dp[n1][n2] = 0;

- when two elements match we just check if the previous elements match and update accordingly

Then we take the max of entire dp {can just be done by keeping a variable and updating while assigning}

- give a single string if we wanna find out the longest palindromic subsequence in it. Just reverse it and create another string. Simply apply LCS on these two !



In order to make a string palindromic, how many min character insertions do you need ?

→ len(s) - len(longest palindromic subsequence)

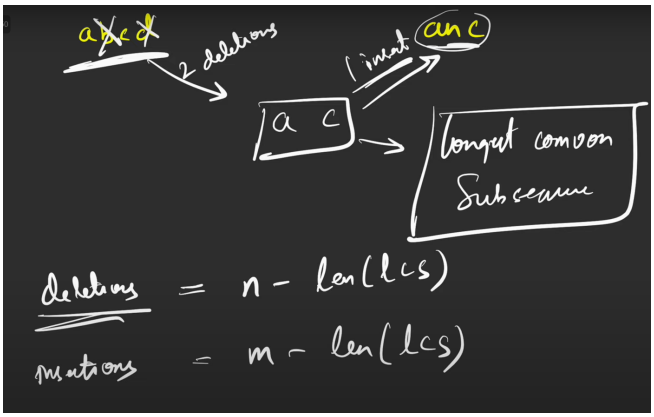
HOW ? see above ex. By keeping LPS constant we just have to adjust the remaining char around to make it palindromic.

MAIN FOCUS IS : **WHAT CAN I NOT TOUCH ?**

The answer will be same if we are asked number of deletions to make a word palindrome

---

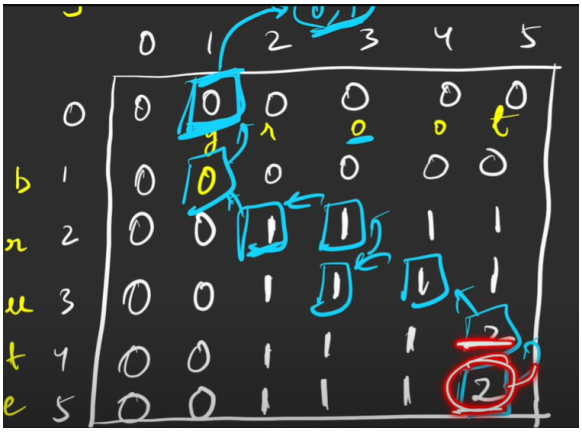
convert a string A to string B. can do insertions and deletions.




```
string ans = "";
int i = n, j = m;
while(i > 0 && j > 0) {
    if(s[i-1] == t[j-1]) {
        ans += s[i-1];
        i--, j--;
    }
    else if(dp[i-1][j] > dp[i][j-1]) {
        ans += s[i-1];
        i--;
    }
    else {
        ans += t[j-1];
        j--;
    }
}
```

```
while(i > 0) {
    ans += s[i-1];
    i--;
}

while(j > 0) {
    ans += t[j-1];
    j--;
}

reverse(ans.begin(), ans.end());

return ans;
```

when we reach the uppermost or left most boundary.



say you have to form the shortest possible superstring, just do LCS then using the dp array we do the above

- Basically we are omitting LCA once {as it is common in both and then arranging remaining non common char in such a way the order remains the same}.

No of distinct subsequences:
Given two strings s and t, how many t are there in s as subsequences ?

```
class Solution {
public:
    int f(int i, int j, string &s, string &t) {
        if(j<0) return 1; // STRING MATCHED
        if(i<0) return 0; // STRING COULD NOT BE MATCHED

        if(s[i] == t[j]) return f(i-1, j-1, s, t) + f(i-1, j, s, t);
        // IF THEY MATCH THEN WE CAN EITHER TAKE THAT OR SKIP IT AND
        // FIND FOR OTHER POSSIBLE MATCHING
```

```
        else return f(i-1, j, s, t); IF THEY DONT MATCH THEN MOVE THE FIRST STRING
    }
    int numDistinct(string s, string t) {
        int m = s.length();
        int n = t.length();
        return f(m-1, n-1, s, t);
    }
};
```

- while converting this into tabulation we have to make dp size {n+1, m+1} so **we can take care of base cases as well**

```
class Solution {

public:
    int f(int i, int j, string &s, string &t, vector<vector<int>> &dp) {
        // base cases

        if(i == 0) return j;
        // S COMPLETED BUT NOT T, JUST ADD REMAINING FIRST CHARACTERS TO S
        if(j == 0) return i;
        // T COMPLETED BUT NOT S, SO JUST DELETE EXTRA
        // CHARACTERS FROM S TO MAKE IT T

        if(dp[i][j] != -1) return dp[i][j];

        // comparision
        if(s[i-1] == t[j-1]) return dp[i][j] = f(i-1, j-1, s, t, dp);
        // if not equal
        else {
            int insert = 1 + f(i, j-1, s, t, dp);
            // ASSUME YOU HAVE ADDED A CHAR IN S AND IT MATCHED,
            // SO YOU SHIFT J AND REMAIN SAME WITH I
            int replace = 1 + f(i-1, j-1, s, t, dp);
            SO YOU REPLACE THEN IT IS SAME CASE AS IF THEY HAVE MATCHED
            int dele = 1 + f(i-1, j, s, t, dp);
            LASTLY IF YOU HAVE DELETED, THEN STAY AT SAME J AND SHIFT BACK I BY 1
            return dp[i][j] = min({insert, replace, dele});
            AS WE NEED MIN NUMBER OF OPERATIONS
        }
    }

    int minDistance(string word1, string word2) {
        int m = word1.length();
        int n = word2.length();
        vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
```

```
        for(int j = 0; j <= n; j++) dp[0][j] = j;
        for(int i = 0; i <= m; i++) dp[i][0] = i;

        for(int i = 1; i <= m; i++) {
            for(int j = 1; j <= n; j++) {
                if(word1[i-1] == word2[j-1]) dp[i][j] = dp[i-1][j-1];
                else {
                    int insert = 1 + dp[i][j-1];
                    int replace = 1 + dp[i-1][j-1];
                    int dele = 1 + dp[i-1][j];
                    dp[i][j] = min({insert, replace, dele});
                }
            }
        }
        return dp[m][n];
    }
};
```

‘*’ and ‘?’ {WILDCARD MATCHING QUESTION}

*→ can be ‘’ or a string with finite length

?→ can be replaced by a character

```
class Solution {
public:
        bool f(int i, int j, string &s, string &p, vector<vector<int>> &dp) {
            // Base cases
            if (i == 0 && j == 0) return true;
            else if (j == 0) return false; CAN DO NOTHING
            else if (i == 0) { CAN DO IF THE REMAINING ARE ALL *
                for (int k = 0; k <= j-1; k++) {
                    if (p[k] != '*') return false;
                }
                return true;
            }

            if (dp[i][j] != -1) return dp[i][j];

            // Comparisons
            if (p[j-1] == '?' || s[i-1] == p[j-1])
                return dp[i][j] = f(i-1, j-1, s, p, dp);
            AS * CAN BE '' JUST MOVE ON {J-1} OR THEY TAKE THE CHAR PRESENT {I-1}
            if (p[j-1] == '*') return dp[i][j] =
                                        f(i-1, j, s, p, dp) || f(i, j-1, s, p, dp);

            return dp[i][j] = false;
        }
```

```
        bool isMatch(string s, string p) {
            int m = s.length();
            int n = p.length();
            vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

                THIS BASE CONVERSION IS VERY TRICKY AND INTERESTING
            for (int i = 1; i <= m; i++) dp[i][0] = 0; // can be omitted
            for (int j = 1; j <= n; j++) {
                int flag = true;
                for (int k = 0; k <= j-1; k++) {
                    if (p[k] != '*') {
                        flag = false;
                        break;
                    }
                }
                if (flag) dp[0][j] = flag;
                else dp[0][j] = 0;
            }
            dp[0][0] = 1;

            for (int i = 1; i <= m; i++) {
                for (int j = 1; j <= n; j++) {
                    if (p[j-1] == '?' || s[i-1] == p[j-1])
                        dp[i][j] = dp[i-1][j-1];
                    else if (p[j-1] == '*')
                        dp[i][j] = dp[i-1][j] || dp[i][j-1];
                    else dp[i][j] = 0;
                }
            }

            return dp[m][n];
        }
};
```

DP on stocks :
The main idea is to keep track of min price on the days before we are selling it and as well as the max profit till now and we are done

```
int maximumProfit(vector<int> &prices) {
    // So I have to keep track of the minimum as I traverse
    int n = prices.size();
    int mini = prices[0];
    int ans = 0;
    // considering you can sell on the same day else use prices[1] - prices[0]

    for (int i = 1; i < n; i++) {
```

```
        int pro = prices[i] - mini;
        ans = max(ans, pro);
        mini = min(mini, prices[i]);
    }
    return ans;
}
```

Now if we can buy any number of stock not operating on twice at a time then

```
long f(int day, int can, long *values, int n, vector<vector<long>> &dp) {
    if (day == n) return 0;
    // IT DOESN'T MATTER YOU CAN'T BUY, SELL SO RETURN 0

    if (dp[day][can] != -1) return dp[day][can];

    if (can == 0) {
        long not_pick = f(day + 1, 0, values, n, dp);
        // CAN BUY BUT NOT TODAY
        long pick = -values[day] + f(day + 1, 1, values, n, dp);
        // BOUGHT SO SPENDING MONEY AND GO TO NEXT WITH THE INTENTIO OF SELLING
        return dp[day][can] = max(not_pick, pick);
    }

    if (can == 1) {
        long sell = values[day] + f(day + 1, 0, values, n, dp);
        // SOLD TODAY AND GOING TO NEXT DAY
        long not_sell = f(day + 1, 1, values, n, dp);
        // KEEPING THE STOCK TO SELL LATER
        return dp[day][can] = max(sell, not_sell);
    }

}

long getMaximumProfit(long *values, int n) {
    // vector<vector<long>> dp(n + 1, vector<long>(2, 0));
    // TO ACCOMODATE BASE CASE
    vector<long> ahead(2, 0); // dp[n][0] = dp[n][1] = 0; {done explicitly}
        // REMEMBER IN RECURSION WE PASSED 0,
        // 0 SO TABULATION WILL HAVE DECREASING ITERRATION
    for (int day = n - 1; day >= 0; day--) {
        vector<long> curr(2, 0);
        for (int can = 0; can <= 1; can++) {
            if (can == 0) {
                long not_pick = ahead[0];
                long pick = -values[day] + ahead[1];
                curr[can] = max(not_pick, pick);
            }
            else {
```

```
                long sell = values[day] + ahead[0];
                long not_sell = ahead[1];
                curr[can] = max(sell, not_sell);
            }
        }
        ahead = curr;
    }
    return ahead[0];
}
```

```
for (int day = n - 1; day >= 0; day--) {
        vector<long> curr(2, 0);
        for (int can = 0; can <= 1; can++) {
            if (can == 0) {
                long not_pick = ahead[0];
                long pick = -values[day] + ahead[1];
                curr[can] = max(not_pick, pick);
            }
            else {
                long sell = values[day] + ahead[0];
                long not_sell = ahead[1];
                curr[can] = max(sell, not_sell);
            }
        }
        ahead = curr;
    }

THIS IS EQUIVALENT TO BELOW ONE AS EITHER OF THEM IS PERFORMED ONCE IN PREVIOUS

for (int day = n - 1; day >= 0; day--) {
        vector<long> curr(2, 0);

        long not_pick = ahead[0];
        long pick = -values[day] + ahead[1];
        curr[0] = max(not_pick, pick);

        long sell = values[day] + ahead[0];
        long not_sell = ahead[1];
        curr[1] = max(sell, not_sell);

        ahead = curr;
    }
```

In case we can only do two transactions {CAP/K = 2} :

```
int f(int day, int can, int cap, vector<int>& prices, int n,
                                                              vector<vector<vector<
```

```
    // either you reach end of the day or you run out of capacity
    if(cap == 0 || day == n) return 0;

    if(dp[day][can][cap] != -1) return dp[day][can][cap];

    if(can == 0) {
        int buy = -prices[day] + f(day+1, 1, cap, prices, n, dp);
        int not_buy = f(day+1, 0, cap, prices, n, dp);
        return dp[day][can][cap] = max(buy, not_buy);
    }
    else {
        int sell = prices[day] + f(day+1, 0, cap-1, prices, n, dp);
        // ONLY AFTER YOU SELL A STOCK THAT YOU BOUGHT THE CAPACITY DECREASES
        int not_sell = f(day+1, 1, cap, prices, n, dp);
        return dp[day][can][cap] = max(sell, not_sell);
    }
}

int maxProfit(vector<int>& prices)
{
    // Need to have three variable : day, buy/sell/skip, how many more ?
    int n = prices.size();
    // vector<vector<vector<int>>> dp(n+1, vector<vector<int>>(2, vector<int>(3, 0)))
    vector<vector<int>> ahead(2, vector<int>(3, 0));
    // cap = 0/1/2 {change accordingly if k is given}
    vector<vector<int>> curr(2, vector<int>(3, 0));

    Repeatedly allocating and deallocating memory,
    especially within tight loops or frequently called functions can lead to
    * Increased Execution Time: The overhead of memory management can add up,
      slowing down the program.
    * whether we declare curr inside or outside
      loop space complexity is still O(1)

    for(int day = n-1;day >= 0;day--) {
        for(int can = 0;can <= 1;can++) {
            for(int cap = 1;cap <= 2;cap++) {
            // BOTH DAY AND CAP ARE INVOLVED IN BASE CASE SO
            // EXCLUDE THAT IN TABULATION
                if(can == 0) {
                    curr[can][cap] =
                        max(-prices[day] + ahead[1][cap], ahead[0][cap]);
                }
                else {
                    curr[can][cap] =
                        max(prices[day] + ahead[0][cap-1], ahead[1][cap]);
                }
```

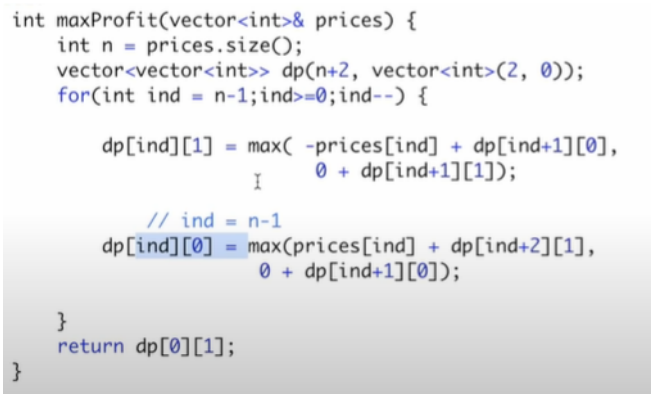
```
            }
        }
        ahead = curr;
    }
    return ahead[0][2];
}
```

The additional space we are using here if O(day+1, can, cap+1) 3D rather we can reshape into 2D with a completely different variable O(day+1, can * (cap+1)) say transNo.

Now the upper limit is 2*k = {can * (cap+1)} AND WHEN THIS IS EVEN YOU CAN BUY ELSE SELL DONE!

---

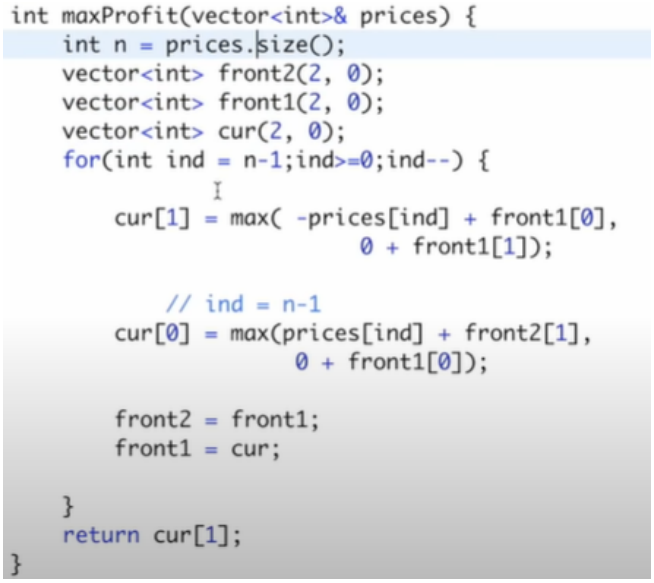
say there was a rule that you cannot buy a stock on the next day if you have sold it today : COOLDOWN



```
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    vector<vector<int>> dp(n+2, vector<int>(2, 0));
    for(int ind = n-1;ind>=0;ind--) {

        dp[ind][1] = max( -prices[ind] + dp[ind+1][0],
                          0 + dp[ind+1][1]);

            // ind = n-1
        dp[ind][0] = max(prices[ind] + dp[ind+2][1],
                         0 + dp[ind+1][0]);

    }
    return dp[0][1];
}
```

- After selling simply shift to day+2

As there are both day+1 and day+2 space optimization here looks like



```
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    vector<int> front2(2, 0);
    vector<int> front1(2, 0);
    vector<int> cur(2, 0);
    for(int ind = n-1;ind>=0;ind--) {

        cur[1] = max( -prices[ind] + front1[0],
                      0 + front1[1]);

            // ind = n-1
        cur[0] = max(prices[ind] + front2[1],
                     0 + front1[0]);

        front2 = front1;
        front1 = cur;

    }
    return cur[1];
}
```

- you can return either cur[1] or front[1] as they are both same {you assign before loop completes}

Lastly say if there were a broker charge for each stock u sell simply subtract that fee whenever you sell and we are done!

Given an array we have to find the length of longest increasing subsequence :

```
class Solution {
public:
        RATHER THAN CARRYING THE PREV MIN NUMBER,
        WE CAN JUST CARRY THEIR INDEX
    int f(int i, int mi_index, int n, vector<int>& nums, vector<vector<int>>& dp) {
        if(i == n) return 0;
        if(dp[i][mi_index + 1] != -1) return dp[i][mi_index + 1];

        int not_take = f(i + 1, mi_index, n, nums, dp);
        int take = 0;
        if(mi_index == -1 || nums[i] > nums[mi_index]) {
        // -1 SUGGETS THERE IS NO MIN YET
            take = 1 + f(i + 1, i, n, nums, dp);
        }
        return dp[i][mi_index + 1] = max(take, not_take);
    }
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> dp(n, vector<int>(n + 1, -1));
        // i CAN GO FORM 0 - n BUT mi_index FROM -1 TO n
        return f(0, -1, n, nums, dp);
    }
};
```

```
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
        WE ARE n+1 TO ACCOMODATE BASE CASE
        // IGNORING BASE CASE AS WE ARE TO ASSIGN 0
        // {already done during initilisation}
        for(int i = n - 1; i >= 0; i--) {
            for(int mi_index = i - 1; mi_index >= -1; mi_index--) {
            REMEMBER HERE mi_index HAS A CONSTRIANT
                int not_take = dp[i + 1][mi_index + 1];
                important here as [-1] is not valid we shift by in 1 in entire dp
                int take = 0;
                if(mi_index == -1 || nums[i] > nums[mi_index]) {
                    take = 1 + dp[i + 1][i + 1]; second parameter in +1 state
                }
                dp[i][mi_index + 1] = max(take, not_take);
            }
        }
```

```
        return dp[0][0]; not -1 as we shifted by 1
    }
```

- It basically is LCS of the array and its sorted array {in increasing order}
- OR else we keep track of only the prev index of a number which will be part of LIS

```
for(int i = 0;i<n;i++) {
    hash[i] = i;
    for(int prev = 0; prev < i; prev++) {
        if(arr[prev] < arr[i] &&
           1 + dp[prev] > dp[i]) {

            dp[i] = 1 + dp[prev];
            hash[i] = prev;
        }
    }
    if(dp[i] > maxi) {
        maxi = dp[i];
        lastIndex = i;
    }
}
```

Here is an algorithmic approach {n*2} for finding LIS :

- as we traverse the array, we keep checking previous elements IF THEY ARE GREATER, and are they a part of LIS {dp keeps track of current LIS} then update the curr i dp to a part of LIS
- the second if, is just to keep track of max dp [last element of LIS] and its length
- the hash[i] will store the index of prev element {SO WE BASICALLY BACKTRACK}

```
vector<int> temp;
temp.push_back(arr[lastIndex]);
while(hash[lastIndex] != lastIndex) {
    lastIndex = hash[lastIndex];
    temp.push_back(arr[lastIndex]);
}
reverse(temp.begin(), temp.end());
```

- the stopping condition basically is the starting element as it points to itself
- as we are storing in the reverse order just ulta it !

GREEDY + BINARY SEARCH APPROACH :

- **Greedy Approach**: By always replacing the element in `temp` with the smallest possible value that can maintain the increasing order, we maximize the chances of extending the subsequence in future iterations.
- **Binary Search**: The use of `lower_bound` ensures that we find the correct position in logarithmic time, making the algorithm efficient.

```
class Solution {
public:
int lengthOfLIS(std::vector<int>& nums) {
    vector<int> temp;
    temp.push_back(nums[0]);
    int len = 1;
        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] > temp.back()) {
                // If current element is greater than the last element in temp,
                // extend the sequence
                temp.push_back(nums[i]);
                len++;
            } else {
                // Otherwise, find the position to replace
                int ind = lower_bound(temp.begin(), temp.end(), nums[i])

- temp.begin();
                temp[ind] = nums[i];
            }
        }
        return len;
    }
};
```

Longest divisible subset :

subset can be any so take the sorted one

```
vector<int> divisibleSet(vector<int> &arr) {
    int n = arr.size();
    sort(arr.begin(), arr.end());

    vector<int> dp(n, 1), hash(n);
    // dp TO TRACK LENGTH OF LCS, hash TO BACKTRACK

    int maxi = 1;
    int last_index = 0;

    for (int i = 0; i < n; i++) {
        hash[i] = i;
        for (int prev = 0; prev < i; prev++) {
            if (arr[i] % arr[prev] == 0 && 1 + dp[prev] > dp[i]) {
            // if 8 is divisible by 4 then it 2 as well
                dp[i] = 1 + dp[prev];
                hash[i] = prev;
            }
```

```
        }
        if (dp[i] > maxi) {
            maxi = dp[i];
            last_index = i;
        }
    }

    vector<int> ans;
    ans.push_back(arr[last_index]);
    while (hash[last_index] != last_index) {
    // THE LAST ELEMENTS GETS LEFT OUT IF NOT LIKE THIS
        last_index = hash[last_index];
        ans.push_back(arr[last_index]);
    }

    reverse(ans.begin(), ans.end());

    return ans;
}
```

Longest String Chain :
take a string, add one character, keep going : LENGHT of biggest chain you made ?

```
bool compare(string &a, string &b) {
    if (a.size() != b.size() + 1) return false;
    // ONLY ONE CHARACTER DIFFERENCE
    int first = 0, second = 0;
    while (first < a.size()) {
        if (second < b.size() && a[first] == b[second]) {
            second++;
        }
        first++;
    }
    return second == b.size();
    // NO NEED TO CHECK FOR first == a.size() CAUSE THAT IS BREAKING CONDITION
}

bool cus_com(string &a, string &b) {
// CUSTOM COMPARATOR : SORT BASED ON LENGTHS OF STRINGS AS WE ONLY NEED
//  A SUBSET NOT SUBSEQUENCE
    return a.size() < b.size();
}

int longestStrChain(vector<string> &arr) {
    int n = arr.size();
    vector<int> dp(n, 1);
    sort(arr.begin(), arr.end(), cus_com);
```

```
    int ans = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (compare(arr[i], arr[j]) && dp[i] < dp[j] + 1) {
            // HERE dp IS NEEDED TO MAKE SURE IT IS STILL
            // PART OF CHAIN AS WE PROGRESS
                dp[i] = dp[j] + 1;
            }
        }
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

Longest Bitonic Subsequence ? → Increasing and then decreasing {CAN BE EITHER ONLY}

→ it is increasing from first + increasing from the last

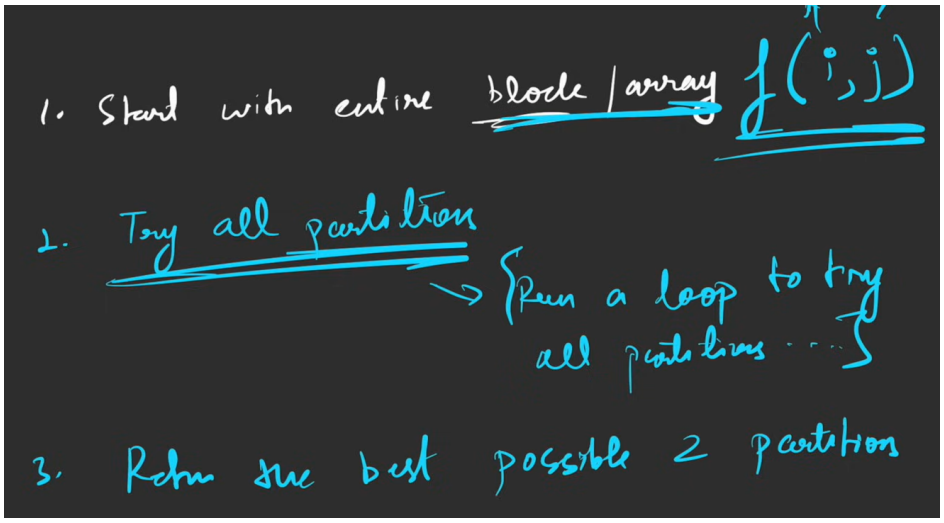
So, perform LIS in normal order and then reverse. Then add the two dp arrays and subtract 1 as there would be a common element.

Number of LIS ?

```
int findNumberOfLIS(vector<int> &arr) {
    int n = arr.size();
    vector<int> dp(n, 1), cnt(n, 1);
    int maxi = 1;

    for (int i = 0; i < n; i++) {
        for (int prev = 0; prev < i; prev++) {
            if (arr[prev] < arr[i] && 1 + dp[prev] > dp[i]) {
            // INHERIT AND CONTINUE WITH LENGTH 1 MORE
                dp[i] = 1 + dp[prev];
                cnt[i] = cnt[prev];
            } else if (arr[prev] < arr[i] && 1 + dp[prev] == dp[i]) {
            // NO NEED TO INHERIT, WE HAVE ALREADY JUST ADD SIMILAR SUCH CASES
                cnt[i] += cnt[prev];
            }
        }
        maxi = max(maxi, dp[i]);
    }

    int res = 0;
    for (int i = 0; i < n; i++) {
        if (dp[i] == maxi) {
        // ENDING ELEMENT WOULD BE DIFFERENT SO ADD THOSE SUCH CASES
            res += cnt[i];
        }
    }
```

```
    return res;
}
```

Dp on PARTITION



Minimum no of steps in Matrix Multiplication :

```
int f(int i, int j, vector<int> &arr, vector<vector<int>> &dp) {
    if(i == j) return 0;
    // SINGLE MATRIX NOTHING TO MULTIPLY
    // Start making partitions
    if(dp[i][j] != -1) return dp[i][j];
    int mini = INT_MAX;
    for(int k = i;k <= j-1;k++) {
    // START MAKING PARTITIONS 1) i->k & 2) k+1->j
        int steps = arr[i-1]*arr[k]*arr[j] + f(i, k, arr, dp) + f(k+1, j, arr, dp);
        // FOR WHATEVER PARTITION IT MAY BE THE FINAL SHAPE WILL
        // BE DECIDED ONLY ON FIRST AND LAST SO (i-1)*j is FIXED
        // while the BETWEEN k CHANGES SO IT WILL BE (i-1*k)*(k*j)
        // => No. of steps = i-1*k*j
        mini = min(steps, mini);
    }
    return dp[i][j] = mini;
}

int matrixMultiplication(vector<int> &arr, int N)
// arr HAS N-1 MATRICES, STARTING FROM 1 -> N, SHAPE IS GIVEN BY arr[i-1]*arr[i];
{
    vector<vector<int>> dp(N, vector<int>(N, -1));
```

```
    return f(1, N-1, arr, dp);
}
```

```
int matrixMultiplication(vector<int> &arr, int N)
{
    vector<vector<int>> dp(N, vector<int>(N, 0));
    // BASE CASE OMITTED AS WE ARE RETURNING 0
    for(int i = N-1;i >= 1;i--) {
    // HAVE TO DO IT REVERSE
        for(int j = i+1;j < N;j++) {
        // j HERE IS DEPENDENT ON i IT IS ALWAYS GREATER
        // {EQUAL TO CASE HAS ALREADY BEEN DEALT IN BASE CASE SO DON'T INCLUDE IT}
            int mini = INT_MAX;
            for(int k = i;k <= j-1;k++) {
                int steps = arr[i-1]*arr[k]*arr[j] + dp[i][k] + dp[k+1][j];
                mini = min(steps, mini);
            }
            dp[i][j] = mini;
        }
    }

    return dp[1][N-1];
}
```

Cut The Sticks : Each cut will cost the length of the stick {min cost ?}

```
int f(int i, int j, vector<int> &cuts, vector<vector<int>> &dp) {
    if(i > j) return 0; NO MEANING

    if(dp[i][j] != -1) return dp[i][j];

    int mini = INT_MAX;
    for(int k = i;k <= j;k++) {
        // EACH TIME WE CUT WE NEED TO ADD THE LENGTH. HOW ?
        // => WE ADD THE START AND ENDING INDICES
        // BUT NEVER USE THEM FOR CUTTING BUT JUST LENGTH CALCULATION.
        // EACH TIME WE MAKE A CUT WE UPDATE THE END INDICES.
        // SO cuts[j+1] - cuts[i-1] gives length.
        int cost = cuts[j+1] - cuts[i-1] + f(i, k-1, cuts, dp) + f(k+1, j, cuts, dp);
        mini = min(mini, cost);
    }
    return dp[i][j] = mini;
}

int minCost(int n, vector<int>& cuts) {
    int c = cuts.size();
```

```
    cuts.push_back(n);
    cuts.insert(cuts.begin(), 0);
    // ADD THE EXTREME INDICES
    sort(cuts.begin(), cuts.end());
    // SORT AS WHEN WE CUT, THE NEXT CUT HAS TO BE IN CORRECT PARTITION
    vector<vector<int>> dp(c+2, vector<int>(c+2, 0));
    // REASON FOR c+2 is WHEN i = 1, 0 index needed or i = n, n+1
    // needed in dp calling : BASE CASE ADJUSTMENT.

    for(int i = c;i >= 1;i--) { REVERSE ORDER
        for(int j = i;j <= c;j++) {
            int mini = INT_MAX;
            for(int k = i;k <= j;k++) {
                int cost = cuts[j+1] - cuts[i-1] + dp[i][k-1] + dp[k+1][j];
                mini = min(mini, cost);
            }
            dp[i][j] = mini;
        }
    }
    return dp[1][c];
}
```

Bursting Balloons :

You burst a balloon you get money = product with adjacent guys. CORNER ? *1

```
int f(int i, int j, vector<int>& arr) {
   if(i > j) return 0;
   int maxi = INT_MIN;
   for(int k = i;k <= j;k++) {
      int money = arr[i-1]*arr[k]*arr[j+1] + f(i, k-1, arr) + f(k+1, j, arr);
      maxi = max(maxi, money);
   }
   return maxi;
}

int maxCoins(vector<int>& a) {
    int n = a.size();
    a.push_back(1);
    a.insert(a.begin(), 1);
    return f(1, n, a);
}
```

REASON WHY THIS WORKS ?

instead of bursting, we start with one balloon and start adding balloons in given order while constantly summing the money we get.

WHY ARE WE DOING IT IN THIS WAY ?

The partitions created during recursion or inter-dependent on each others elements.

SO WHEN TO APPLY THIS?

when inter-dependency is seen.