Dynamic Programming

Dynamic Programming (DP) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions for later looking up instead of recomputation. DP has two key attributes : optimal substructure and overlapping sub-problems. If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called “divide and conquer” instead.

Steps

- Find recursive variable dp Setup border condition

$$dp[i] = init$$

- Find recursive rules

$$dp[i] = f([dp[j]]_{j=0..i-1})$$

- Iteration and apply rules
  - Top-down
  - Bottom-up

Two approaches:

- **Top-down approach** We define a solution table. We look up the table for solutions, if not recorded we compute and save the its solution to the table. compute it otherwise.

- **Bottom-up approach** We solve the first and use their solutions to build-on and arrive at solutions to bigger sub-problems.

Typical problems

The DP problem can be summarized into the following types.

1. k depth recursion

In this type problems, the DP rule is simple. $dp[i]$ can be expressed as a simple function of k number of previous solutions. The time complexity is $O(n)$ for 1D and $O(mn)$ for 2D. The space complexity is the same, but can be further reduced to $O(1)$, since we just need k previous solutions. We could use $dp1, dp2, ..dpk$ to replace $dp[0...n-1]$.

$$dp[i] = f(dp[i-1], dp[i-1], .., dp[i-k])$$

For example in 62.Unique Paths, the dp variable is Fibonacci numbers, and rule is $dp[i] = dp[i-1] + dp[i-2]$.

1D

70.Climbing Stairs

```
int climbStairs(int n) {
vector<int> dp={1,2};
 for(int i=2;i<n;i++)dp.push_back(dp[i-2]+dp[i-1]);
 return dp[n-1];
}
```

91.Decode Ways

```
int numDecodings(string s) {
  int n = s.size();
  if(!n || s[0] == '0')
      return 0;
  int f[n+1] = {1, 1};
  for( int i = 2; i <= n; ++i)
       f[i] = (s[i-1] != '0')*f[i-1] + ((s[i-2] == '1') || (s[i-2] == '2' && s[i-1] <=
'6'))*f[i-2];
return f[n];
}
```

639. Decode Ways II

```
int numDecodings(string s) {
    long e0 = 1, e1 = 0, e2 = 0, f0 = 0, M = 1e9 + 7;
    for (char c : s) {
        if (c == '*') {
            f0 = 9 * e0 + 9 * e1 + 6 * e2;
            e1 = e0;
            e2 = e0;
        } else {
            f0 = (c > '0') * e0 + e1 + (c <= '6') * e2;
            e1 = (c == '1') * e0;
            e2 = (c == '2') * e0;
        }
        e0 = f0 % M;
    }
    return e0;
}
```

198.House Robber

```
int rob(vector<int>& nums) {
   if(nums.empty())return 0;
    int n = nums.size();
     vector<int> dp={0,nums[0]};
    for(int i=2;i<=n;++i){
        dp.push_back(max(dp[i-1], nums[i-1]+dp[i-2]));
    }

    return dp[n];
}
```

213.House Robber II

```
int rob(vector<int>& nums) {
   if(nums.empty())return 0;
    int n = nums.size();
     if(n==1) return nums[0];
    vector<int> dp1={0,nums[0]}, dp2={nums[n-1],nums[n-1]};
    for(int i=2;i< n;++i){
        dp1.push_back(max(dp1[i-1], nums[i-1]+dp1[i-2]));
        dp2.push_back(max(dp2[i-1], nums[i-1]+dp2[i-2]));
    }

    return max( dp1[n-1], dp2[n-2]);
}
```

2D 62. Unique Paths

```
int uniquePaths(int m, int n) {
 vector<vector<int>>path(m, vector<int>(n,1));

 for(int i=1;i<m;++i){
     for(int j=1;j<n;++j){
         path[i][j]=path[i-1][j]+path[i][j-1];
     }
 }
 return path[m-1][n-1];
 }
```

63.Unique Paths II

```
int uniquePathsWithObstacles(vector<vector<int>>& grid) {
    int m=grid.size(), n=grid[0].size();
```

```
    vector<vector<int>>path(m, vector<int>(n,0));
    if(grid[m-1][n-1]==1) return 0;
     path[m-1][n-1]=1;

  for(int i=m-2;i>=0;i--){
        path[i][n-1]=grid[i][n-1]==1? 0:  path[i+1][n-1];
    }
    for(int j=n-2;j>=0;j--){
          path[m-1][j]=grid[m-1][j]==1? 0:  path[m-1][j+1];
    }
   for(int i=m-2;i>=0;--i){
      for(int j=n-2;j>=0;--j){
          path[i][j]= grid[i][j]==1? 0:  path[i+1][j]+path[i][j+1];
      }
  }
return path[0][0];
}
```

64. Minimum Path Sum

```
int minPathSum(vector<vector<int>>& grid) {
    int m=grid.size(), n= grid[0].size();

    for(int i=m-2;i>=0;--i ) grid[i][n-1]= grid[i][n-1] + grid[i+1][n-1];
    for(int j=n-2;j>=0;--j ) grid[m-1][j]= grid[m-1][j] + grid[m-1][j+1];


    for(int i=m-2;i>=0;--i){
        for(int j=n-2;j>=0;--j){
            grid[i][j]=grid[i][j]+min(grid[i+1][j],grid[i][j+1]);
        }

    }
    return grid[0][0];
}
```

3. Whole path recursion

In this type problems, $dp[i]$ is defined as a function of all previous soltion. a simple function of k number of previous solutions. The time complexity is $O(n^2)$ for 1D. The space complexity is $O(n)$.

95. Unique Binary Search Trees

```
int numTrees(int n) {
    vector<int>dp(n+1,0);
    dp[0]=1;

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

```
        for(int j=0;j<i;++j)
            dp[i]+=dp[j]*dp[i-j-1];
    }

   return dp[n];
}
```

647.Palindromic Substrings

```
int countSubstrings(string s) {
    int n =s.size();
    vector<int>dp(n+1,0);
     vector<vector<bool>> m(n, vector<bool>(n, false));
    for(int i = n-1; i>=0; --i){
        dp[i]+=dp[i+1];
        for(int j = i; j<n;++j){
            if(i==j || (s[i]==s[j]&& (m[i+1][j-1]||j==i+1))){
                m[i][j]=true;
                dp[i]+= 1;
             }
        }

    }
    return dp[0];

}
```

132. Palindrome Partitioning II Two dp variable dp and m are running here.

```
int minCut(string s) {
    int n=s.size();
    vector<int> dp;
    for(int i=0; i<=n; ++i) dp.push_back(n-i-1);
    vector<vector<bool>> m(n, vector<bool>(n, false));
    for(int i = n-1; i>=0; --i){
        for(int j = i; j<n;++j){
            if(i==j || (s[i]==s[j]&& (m[i+1][j-1]||j==i+1))){
                m[i][j]=true;
                dp[i]=min(dp[i], 1+dp[j+1]);
            }
        }
    }
    return dp[0];
}
```

Iterate by length of substrings. 516. Longest Palindromic Subsequence

```
int longestPalindromeSubseq(string s) {
    int n =s.size(), ans=0;
    vector<vector<int>>dp(n,vector<int>(n,0));
    for(int i=0; i<n; ++i) dp[i][i]=1;
    for(int len = 2; len <= n; ++len){ //traverse the length
        for(int i = 0; i <= n - len; ++i){
            int j = i + len -1;
            if(s[i]==s[j]) dp[i][j]=dp[i+1][j-1]+2;
            else dp[i][j]= max(dp[i+1][j], dp[i][j-1]);
         }
       }
    return dp[0][n-1];
}
```

730. Count Different Palindromic Subsequences

```
int countPalindromicSubsequences(const string& s) {

    static constexpr long mod = 1000000007;
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) dp[i][i] = 1;
    for (int len = 1; len <= n; ++len) {
        for (int i = 0; i < n - len; ++i) {
            const int j = i + len;
            if (s[i] == s[j]) {
                dp[i][j] = dp[i + 1][j - 1] * 2;
                int l = i + 1, r = j - 1;
                while (l <= r && s[l] != s[i]) ++l;
                while (l <= r && s[r] != s[i]) --r;
                if (l == r) dp[i][j] += 1;
                else if (l > r) dp[i][j] += 2;
                else dp[i][j] -= dp[l + 1][r - 1];
            } else {
                dp[i][j] = dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1];
            }

            dp[i][j] = (dp[i][j] + mod) % mod;
        }
    }
    return dp[0][n - 1];
}
```

4. Global local maximization

The type of problem involves two dp. One is to find local maximum by n-depth or whole path recursion. The other is find the global maximum from local maximum.

53.Maximum Subarray

```
int maxSubArray(vector<int>& nums) {

    int sum= 0, ans= INT_MIN;
    for (auto x:nums){
         sum = max(sum+x, x);
         ans = max(sum, ans);
    }
    return ans;
}
```

152.Maximum Product Subarray

```
int maxProduct(vector<int>& nums) {
    if(nums.empty()) return 0;
    int ans, curMax, curMin;
    ans = curMax = curMin = nums[0];

    for(int i=1;i<nums.size();++i) {
        int x=nums[i];
        if(x<0) swap(curMin, curMax);
        curMax = max(x, x*curMax);
        curMin = min(x, x*curMin);
        ans = max(ans, curMax);
    }
    return ans;
}
```

300.Longest Increasing Subsequence

```
int lengthOfLIS(vector<int>& nums) {
    if(nums.empty()) return 0;
    int n = nums.size();
   vector<int> dp(n,1);
    for(int i=1;i<n;++i)
        for(int j=0;j<i;++j)
            if(nums[i]>nums[j])
                dp[i]=max(dp[i],dp[j]+1);
    return *max_element(dp.begin(),dp.end());
}
```

Using push new element to the back of the answer array if increasing or replace the lower_bound.

```
int lengthOfLIS(vector<int>& nums) {
    vector<int> dp;
    for (int x : nums) {
        auto ix = lower_bound (dp.begin(), dp.end(), x);
        if (ix == dp.end()) dp.push_back(x);
        else *ix = x;
    }
    return dp.size();
}
```

368.Largest Divisible Subset

```
vector<int> largestDivisibleSubset(vector<int>& nums) {
   int n = nums.size(), left=0, len=0;
   vector<int> dp(n,0),pre(n, 0), ans;
    sort(nums.begin(), nums.end());
    for(int i=n-1;i>=0; --i)
        for(int j=i;j< n;++j)
            if(nums[j]%nums[i]==0 && dp[j]+1>dp[i]){
                dp[i] = dp[j]+1,pre[i] = j;
                if(dp[i] > len) len = dp[i], left=i;            }

   for(int i=0; i< len; ++i) ans.push_back(nums[left]), left=pre[left];
    return ans;
}
```

32.Longest Valid Parentheses

```
int longestValidParentheses(string s) {
    int n =s.size(), ans =0;
    vector<int> dp(n+1,0);
    for (int i = 2; i <= n; i++) {
            if (s[i-1] == ')') {
                if (s[i-2] == '(') dp[i] = dp[i-2] +2;
                else {
                    int start =i-2-dp[i-1];
                    if (s[start] == '(')  dp[i] = dp[i-1] + 2 + dp[start];
              }
        }
        ans = max(ans,dp[i]);
    }

    return ans;
  }
```

Alternate solution: using stack

```
int longestValidParentheses(string s) {
    stack<int> stk;   // positions first: index, second: 0:'(', 1:')'
    int maxLen = 0, last=-1;


    for(int i=0; i<s.size(); i++) {
        if(s[i]=='(') stk.push(i);   // left parenthesis
        else if (s[i] == ')') {          // right parenthesis
            if(stk.empty()) last=i;
            else {
                stk.pop();

                int cnt = stk.empty()? i-last: i-stk.top();
                maxLen = max(maxLen, cnt);
            }
        }
    }
```

5. Between two Subsequences

This type of problem usually involves Transformation a string s1 to another s2. The $dp[i][j]$ is defined as a function of subsequence $s1[0...i-1]$ and $s2[0...i-1]$. The update of $dp[i][j]$ depends on comparison between $s1[i-1]$ and $s2[i-1]$. dp is a 2-D array. The time complexity is $O(n^2)$ and the space complexity is $O(n^2)$.

712. Minimum ASCII Delete Sum for Two Strings

```
int minimumDeleteSum(string s1, string s2) {
      int n1 = s1.size(), n2 = s2.size();
    vector<vector<int>> dp(n1+1, vector<int>(n2+1,0));
    for(int i=1; i<=n1;++i) dp[i][0] = s1[i-1]+dp[i-1][0];
    for(int i=1; i<=n2;++i) dp[0][i] = s2[i-1]+dp[0][i-1];
    for(int i=1; i<=n1; ++i){
        for(int j=1; j<=n2; ++j){
            if(s1[i-1]==s2[j-1]) dp[i][j] = dp[i-1][j-1];
            else dp[i][j]=  min(dp[i][j-1 ]+s2[j-1],dp[i - 1][j]+s1[i-1]);
        }
    }
    return dp[n1][n2];
}
```

583. Delete Operation for Two Strings

```
int minDistance(string s1, string s2) {
    int n1 = s1.size(), n2 = s2.size();
    vector<vector<int>> dp(n1+1, vector<int>(n2+1,0));
    for(int i=1; i<=n1;++i) dp[i][0] = i;
```

```
        for(int i=1; i<=n2;++i) dp[0][i] = i;
        for(int i=1; i<=n1; ++i){
            for(int j=1; j<=n2; ++j){
                if(s1[i-1]==s2[j-1]) dp[i][j] = dp[i-1][j-1];
                else dp[i][j]=  min(dp[i][j-1 ],dp[i - 1][j])+1;
            }
        }
        return dp[n1][n2];
    }
```

72. Edit Distance

```
    int minDistance(string word1, string word2) {
        int n1 = word1.size(), n2 = word2.size();
        vector<vector<int>> dp(n1+1, vector<int>(n2+1,0));
        for(int i=0; i<=n1;++i) dp[i][0] = i;
        for(int i=0; i<=n2;++i) dp[0][i] = i;
        for(int i=1; i<=n1; ++i){
            for(int j=1; j<=n2; ++j){
                if(word1[i-1]==word2[j-1]) dp[i][j] = dp[i-1][j-1];
                else dp[i][j]=  min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) +
1;
            }
        }
        return dp[n1][n2];
    }
```

115. Distinct Subsequences

```
    int numDistinct(string s2, string s1) {
         int n1 = s1.size(), n2 = s2.size();
        vector<vector<int>> dp(n1+1, vector<int>(n2+1,0));
        for(int i=0; i<=n2;++i) dp[0][i] = 1;
        for(int i=1; i<=n1; ++i){
            for(int j=1; j<=n2; ++j){
                if(s1[i-1]==s2[j-1]) dp[i][j] =dp[i][j-1 ]+dp[i - 1][j - 1];
                else dp[i][j]= dp[i][j - 1];
            }
        }
        return dp[n1][n2];
    }
```

97. Interleaving String

```
   bool isInterleave(string s1, string s2, string s3) {
        int n1 = s1.size(), n2 = s2.size(), n3 = s3.size();
```

```
        if(n3 != n1+n2) return false;
        vector<vector<bool>> dp(n1+1, vector<bool>(n2+1,true));
        for(int i=1; i<=n1;++i) dp[i][0] = dp[i-1][0]&&s1[i-1]==s3[i-1];
        for(int i=1; i<=n2;++i) dp[0][i] = dp[0][i-1]&&s2[i-1]==s3[i-1];
        for(int i=1; i<=n1; ++i){
            for(int j=1; j<=n2; ++j){
                dp[i][j]= dp[i-1][j]&&s1[i-1] == s3[i+j-1] ||dp[i][j-1]&&s2[j-1] ==
s3[i+j-1];
            }
        }
        return dp[n1][n2];
    }
```

6. Break the subsequence

This type of problem usually involves collect rewards for some action at the element of sequence, which will reduce the sequence. The target is to maximizing the rewards. The $dp[i][j]$ is defined on a subsequence from ith to jth. We need another iteration from i to j to find the local minimal. The time complexity is $O(n^3)$.

312. Burst Balloons

$$dp[i][j] = \max_{k=i..j}([dp[i][k-1] + dp[k+1][j] + f[k])$$

```
int maxCoins(vector<int>& nums) {

     nums.insert(nums.begin(), 1), nums.push_back(1);   // add 1 at both first and last
for convenience
    int n = nums.size();
    vector<vector<int>> dp(n, vector<int>(n,0));// dp[l][r] is the max result of nums[l -
r] inclusively
    for (int j = 1; j < n-1; j++)
        for (int i = j; i > 0; i--)
            for (int k = i; k <= j; k++)            // treat k as the last balloons
bursted
                dp[i][j] = max(dp[i][j], dp[i][k - 1] + nums[i - 1] * nums[k] * nums[j +
1] + dp[k + 1][j]);
    return dp[1][n - 2];

}
```

546. Remove Boxes

Top down solution

```
int removeBoxes(vector<int>& s) {
   if(s.empty()) return 0;
```

```
    int dp[100][100][100] = {0};
    function<int(int, int , int)> dfs =[&](int i, int j, int k){
        if (i > j) return 0;
        if (dp[i][j][k]) return dp[i][j][k];
        while(j>i&&s[j]==s[j-1]) j--,k++;

        dp[i][j][k] = dfs(i, j - 1, 0) + (1 + k) * (1 + k);
        for (int pos = i; pos < j; pos++) {
            if (s[pos] == s[j]) dp[i][j][k] = max(dp[i][j][k], dfs(i, pos, k+1) + dfs(pos
+ 1, j - 1, 0));
        }
        return dp[i][j][k];
    };
    return dfs(0, s.size()-1, 0);
}
```

664. Strange Printer

Topdown solution

```
int strangePrinter(string s) {
     if(s.empty()) return 0;
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n,0));

    function<int(int, int)> dfs =[&](int i, int j){
        if (i > j) return 0;
         if (dp[i][j]) return dp[i][j];
        dp[i][j] = dfs(i, j - 1)+1;
        for (int pos = i; pos < j; pos++) {
            if (s[pos] == s[j]) dp[i][j]= min(dp[i][j], dfs(i, pos) + dfs(pos+1, j-1));
        }
        return dp[i][j];
    };

  return dfs(0, n-1);
}
```