DP

- **Kadane's Algorithm (Maximum subarray sum) (with getting the max subarray also) :-**

```
int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0,
       start =0, end = 0, s=0;

    for (int i=0; i< size; i++ )
    {
        max_ending_here += a[i];
        if (max_so_far < max_ending_here)
        {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }
        if (max_ending_here < 0)
        {
            max_ending_here = 0;
            s = i + 1;
        }
    }
    return (end - start + 1);
}
```

- **Longest Common Subsequence (LCS) :-**
  - Printing ---

```
Make 2D dp matrix
i = m, j = n
while(i > 0 && j > 0)
      if(X[i-1] == Y[j-1])
            Lcs[index--] = X[i-1]
            I--; j--;
      Else if(dp[i-1][j] > d[i][j-1])
            I--;
      Else
            j--;
```

https://www.geeksforgeeks.org/printing-longest-common-subsequence/

  - Space optimised :-

```
int lcs(string &X, string &Y)
{
    int m = X.length(), n = Y.length();
    int L[2][n + 1];
    bool bi;
    for (int i = 0; i <= m; i++)
    {
        bi = i & 1;
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                L[bi][j] = 0;
```

```
                else if (X[i-1] == Y[j-1])
                     L[bi][j] = L[1 - bi][j - 1] + 1;
                else
                    L[bi][j] = max(L[1 - bi][j], L[bi][j - 1]);
            }
        }
        return L[bi][n];
    }
```

- **LCS (Longest Common Subsequence) of three strings :-**

  The idea is to take a 3D array to store the length of common subsequence in all 3 given sequences i. e., L[m + 1][n + 1][o + 1]

  1- If any of the string is empty then there is no common subsequence at all then

  ```
  L[i][j][k] = 0
  ```

  2- If the characters of all sequences match (or X[i] == Y[j] ==Z[k]) then

  ```
  L[i][j][k] = 1 + L[i-1][j-1][k-1]
  ```

  3- If the characters of both sequences do not match (or X[i] != Y[j] || X[i] != Z[k] || Y[j] !=Z[k]) then

  ```
  L[i][j][k] = max(L[i-1][j][k],
                   L[i][j-1][k],
                   L[i][j][k-1])
  ```

  Can't do lcs(lcs(a,b), c)

- **Longest Increasing Subsequence (LIS) :-**
  - O(N*N)

    ```
    Int dp = [1];
    for(int i = 1; i < n; i++)
        for(int j = 0; j < i; j++)
            if(arr[j] < arr[i])
                Dp[i] = max ( dp[i] , dp[j] + 1);
    ```

    Construction :-
    https://www.geeksforgeeks.org/construction-of-longest-increasing-subsequence-using-dynamic-programming/

  - O(NlogN)

    1. If A[i] is smallest among all *end* candidates of active lists, we will *start* new active list of length 1.

    2. If A[i] is largest among all *end* candidates of active lists, we will clone the *largest* active list, and extend it by A[i].

    3. If A[i] is in between, we will find a list with *largest end element that is smaller than* A[i].

```
Clone and extend this list by A[i]. We will discard all
other lists of same length as that of this modified list.
```

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/
Construction :-
https://www.geeksforgeeks.org/construction-of-longest-monotonically-increasing-subsequence-n-log-n/

- Printing :-

```
vector<vector<int> > L[n];
L[0].push_back(arr[0])
for(int i = 1; i < n; i++)
{
      for(int j = 0; j < i; j++)
      {
            if(arr[j] < arr[i] && L[j].size() > L[i].size())
            // reason of not >= is because L[i] will be empty or will contain
            arr[i]
            {
                  L[i] = L[j];
            }
      }
      L[i].push_back(arr[i]);
}
```

- **Longest Common Increasing Subsequence :-**
https://www.geeksforgeeks.org/longest-common-increasing-subsequence-lcs-lis/
The idea is to use dynamic programming here as well. We store the longest common increasing sub-sequence ending at each index of arr2[]. We create an auxiliary array table[] such that table[j] stores length of LCIS ending with arr2[j]. At the end, we return maximum value from this table. For filling values in this table, we traverse all elements of arr1[] and for every element arr1[i], we traverse all elements of arr2[]. If we find a match, we update table[j] with length of current LCIS. To maintain current LCIS, we keep checking valid table[j] values.

For every element in arr1 present in arr2 also, we mark table[j] by current if current is greater (LCS).
While traversing arr2, if we find a smaller element than arr1[i], then we change current to propagate the increased current value for arr1[i] == arr2[j]

```
int LCIS(int arr1[], int n, int arr2[], int m)
{
    int table[m];
    for (int j=0; j<m; j++)
        table[j] = 0;
    for (int i=0; i<n; i++)
    {
        int current = 0;
        for (int j=0; j<m; j++)
        {
            if (arr1[i] == arr2[j])
                if (current + 1 > table[j])
```

```
                    table[j] = current + 1;
            if (arr1[i] > arr2[j])
                if (table[j] > current)
                    current = table[j];
        }
    }
    int result = 0;
    for (int i=0; i<m; i++)
        if (table[i] > result)
           result = table[i];
    return result;
}
```

-> current denotes the number of elements less than arr1[i] (also common as table[j] will be updated for common elements only) in arr2

Example :- 2,1,2 and 2,1,2,2
1st iteration :- arr1[i] = 2
Table = 1, 0, 1, 1
2nd iteration :- arr1[i] = 1
Table = 1, 1, 1, 1
3rd Iteration :- arr1[i] = 3
Table = 1, **1**, 2, 2 (found greater value than current at j where arr2[j] < arr1[i])

- **Longest bitonic Sequence (LBS) :-**
  - O(n*n)
    Calculate LIS and LDS(Longest Decreasing subsequence) for every index.
    Take max of LIS[i] + LDS[i] - 1;

  - Printing
    LIS[0] = {arr[O]}
    LIS[i] = {Max(LIS[j])} + arr[i] where j < i and arr[j] < arr[i]
           = arr[i], if there is no such j

    LDS[n] = {arr[n]}
    LDS[i] = arr[i] + {Max(LDS[j])} where j > i and arr[j] < arr[i]
           = arr[i], if there is no such j
    reverse (LDS[i])

```
    for (int i = 1; i < n; i++)
        {
            for (int j = 0; j < i; j++)
            {
                if ((arr[j] < arr[i]) &&
                    (LIS[j].size() > LIS[i].size()))
                    LIS[i] = LIS[j];
            }
            LIS[i].push_back(arr[i]);
        }
```

- **Longest Bitonic Subsequence in O(n log n) :-**

```
int getLBSLengthLogn(int arr[], int n)
{
    if (n == 0)
        return 0;
    int increasing[n];
    int tail1[n];
    int decreasing[n];
    int tail2[n];
    increasing[0] = arr[0];
    int in = 1;
    tail1[0] = 0;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] < increasing[0])
        {
            increasing[0] = arr[i];
            tail1[i] = 0;
        }
        else if (arr[i] > increasing[in - 1])
        {
            increasing[in++] = arr[i];
            tail1[i] = in - 1;
        }
        else
        {
            increasing[ceilIndex(increasing, -1,in - 1, arr[i])] = arr[i];
            tail1[i] = ceilIndex(increasing, -1,in - 1, arr[i]);
        }
    }
    in = 1;

    revereseArr(arr, n);
    decreasing[0] = arr[0];
    tail2[0] = 0;

    for (int i = 1; i < n; i++)
    {
        if (arr[i] < decreasing[0])
        {
            decreasing[0] = arr[i];
            tail2[i] = 0;
        }
        else if (arr[i] > decreasing[in - 1])
        {
            decreasing[in++] = arr[i];
            tail2[i] = in - 1;
        }
        else
        {
            decreasing[ceilIndex(decreasing, -1, in - 1, arr[i])] = arr[i];
            tail2[i] = ceilIndex(decreasing, -1,in - 1, arr[i]);
```

```
        }
    }
    revereseArr(arr, n);
    revereseArr(tail2, n);
    int ans = 0;
    for (int i = 0; i < n; i++)
        if (ans < (tail1[i] + tail2[i] + 1))
            ans = (tail1[i] + tail2[i] + 1);
    return ans;
}
```

- **Longest Repeating Subsequence**
  Similar to LCS, find the LCS(str, str)where str is the input string with the restriction that when both the characters are same, they shouldn't be on the same index in the two strings.
  https://www.geeksforgeeks.org/longest-repeating-subsequence/

- **Longest Palindrome Subsequence :-**
  - Using LCS and Printing
    Make a reverse string.
    find LCS == LPS

  - 2D matrix (nxn)
    a[i][i] = 1;
    if(s[i] == s[j]) a[i][j] = 2 + a[i+1][j-1];
    if (i+1 == j && s[i] == s[j]) a[i][j] = 2;
    else (s[i] != s[j]) a[i][j] == max(a[i+1][j], a[i][j-1])

  - Array (nx1) :-
    https://www.geeksforgeeks.org/longest-palindrome-subsequence-space/

- **Count All Palindromic Subsequence in a given String :-**
  Initial Values : i= 0, j= n-1;

```
CountPS(i,j)
// Every single character of a string is a palindrome
// subsequence
if i == j
   return 1 // palindrome of length 1

// If first and last characters are same, then we
// consider it as palindrome subsequence and check
// for the rest subsequence (i+1, j), (i, j-1)
Else if (str[i] == str[j])]
   return   countPS(i+1, j) + countPS(i, j-1) + 1;

else
   // check for rest sub-sequence and  remove common
   // palindromic subsequences as they are counted
   // twice when we do countPS(i+1, j) + countPS(i,j-1)
   return countPS(i+1, j) + countPS(i, j-1) - countPS(i+1, j-1)
```

- **The painter's partition problem :-**
  We have to paint n boards of length {A1, A2…An}. There are k painters available and each takes 1 unit time to paint 1 unit of board. The problem is to find the minimum time to get this job done under the constraints that any painter will only paint continuous sections of boards, say board {2, 3, 4} or only board {1} or nothing but not board {2, 4, 5}.

```
int findMax(int arr[], int n, int k)
{
    int dp[k + 1][n + 1] = { 0 };

    for (int i = 1; i <= n; i++)
        dp[1][i] = sum(arr, 0, i - 1);

    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {
            int best = INT_MAX;
            for (int p = 1; p <= j; p++)
                best = min(best, max(dp[i - 1][p],
                                     sum(arr, p, j - 1)));

            dp[i][j] = best;
        }
    }
    return dp[k][n];
}
```

  Optimizations:
  1) The time complexity of the above program is O(k*N^3). It can be easily brought down to O(k*N^2) by precomputing the cumulative sums in an array thus avoiding repeated calls to the sum function:

```
 int sum[n+1] = {0};
 for (int i = 1; i <= n; i++)
   sum[i] = sum[i-1] + arr[i-1];

 for (int i = 1; i <= n; i++)
   dp[1][i] = sum[i];
```

  and using it to calculate the result as:
  `best = min(best, max(dp[i-1][p], sum[j] - sum[p]));`

  2) Though here we consider to divide A into k or fewer partitions, we can observe that
  the optimal case always occurs when we divide A into exactly k partitions. So we can use:

```
for (int i = k-1; i <= n; i++)
    best = min(best, max( partition(arr, i, k-1), sum(arr, i, n-1)));
```

  and modify the other implementations accordingly.

- **The painter's partition problem O(N*log(sum(arr[])) :-**
  We can see that the highest possible value in this range is the sum of all the elements in the array and this happens when we allot 1 painter all the sections of the board. The lowest possible value of this range is the maximum value of the array max, as in this allocation we can allot max to one painter and divide the other sections such
  that the cost of them is less than or equal to max and as close as possible to max. Now if we consider we use x painters in the above scenarios, it is obvious that as the value in the range increases, the value of x decreases and vice-versa. From this we can find the target value when x=k and use a helper function to find x, the minimum number of painters required when the maximum length of section a painter can paint is given.

```
int numberOfPainters(int arr[], int n, int maxLen)
{
    int total = 0, numPainters = 1;
    for (int i = 0; i < n; i++) {
        total += arr[i];
        if (total > maxLen) {
            total = arr[i];
            numPainters++;
        }
    }
    return numPainters;
}
int partition(int arr[], int n, int k)
{
    int lo = getMax(arr, n);
    int hi = getSum(arr, n);

    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        int requiredPainters = numberOfPainters(arr, n, mid);
        if (requiredPainters <= k)
            hi = mid;
        else
            lo = mid + 1;
    }
    return lo;
}
```

- **Box Stacking Problem** :- You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

  Following are the key points to note in the problem statement:
  1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.

2) We can rotate boxes such that width is smaller than depth. For example, if there is a box with dimensions {1x2x3} where 1 is height, 2×3 is base, then there can be three possibilities, {1x2x3}, {2x1x3} and {3x1x2}
3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

Following is the solution based on DP solution of LIS problem.

1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.

2) Sort the above generated 3n boxes in decreasing order of base area.

3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.
MSH(i) = Maximum possible Stack Height with box i at top of stack
MSH(i) = { Max (MSH(j)) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i).
If there is no such j then MSH(i) = height(i)

4) To get overall maximum height, we return max(MSH(i)) where 0 < i < n

```
int maxStackHeight( Box arr[], int n )
{
   Box rot[3*n];
   int index = 0;
   for (int i = 0; i < n; i++)
   {
      rot[index].h = arr[i].h;
      rot[index].d = max(arr[i].d, arr[i].w);
      rot[index].w = min(arr[i].d, arr[i].w);
      index++;

      rot[index].h = arr[i].w;
      rot[index].d = max(arr[i].h, arr[i].d);
      rot[index].w = min(arr[i].h, arr[i].d);
      index++;

      rot[index].h = arr[i].d;
      rot[index].d = max(arr[i].h, arr[i].w);
      rot[index].w = min(arr[i].h, arr[i].w);
      index++;
   }
   n = 3*n;
   qsort (rot, n, sizeof(rot[0]), compare);

   int msh[n];
   for (int i = 0; i < n; i++ )
      msh[i] = rot[i].h;

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

```
        for (int j = 0; j < i; j++ )
           if ( rot[i].w < rot[j].w &&
                rot[i].d < rot[j].d &&
                msh[i] < msh[j] + rot[i].h
              )
           {
                msh[i] = msh[j] + rot[i].h;
           }
     int max = -1;
     for ( int i = 0; i < n; i++ )
        if ( max < msh[i] )
           max = msh[i];
     return max;
  }
```

- **Matrix Chain Multiplication :-**
  - O(n*n*n)

```
    i = 1, j = n-1
    for (k = i; k < j; k++)
        {
            count = MatrixChainOrder(p, i, k) +
                    MatrixChainOrder(p, k + 1, j) +
                    p[i - 1] * p[k] * p[j];

            if (count < min)
                min = count;
        }
```

    https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/

  - O(n*n)
    Sometimes Wrong also

    Assume there are following available method
    minCost(M1, M2) -> returns min cost of multiplying matrices M1 and M2

    Then, for any chained product of matrices like,

    M1.M2.M3.M4…Mn

    min cost of chain = min(minCost(M1, M2.M3…Mn), minCost(M1.M2..Mn-1, Mn))

    Now we have two subchains (sub problems) :

    M2.M3…Mn

    M1.M2..Mn-1, Mn

    https://www.geeksforgeeks.org/matrix-chain-multiplication-a-on2-solution/

  - Printing - space O(n*n)
    Bracket matrix stores the k value for each i,j
    printParenthesis(i, j, bracket[n][n], name)
    {

```
        if (i == j)
        {
            print name;
            name++;
            return;
        }

        print "(";
        printParenthesis(i, bracket[i][j], bracket, name);
        printParenthesis(bracket[i][j]+1, j, bracket, name);

        print ")";
    }
```

https://www.geeksforgeeks.org/printing-brackets-matrix-chain-multiplication-problem/

- Printing space optimised :-
  Bracket value k for each i,j can be stored in dp[j][i] as it is unused
  https://www.geeksforgeeks.org/printing-matrix-chain-multiplication-a-space-optimized-solution/

- **Minimum number of jumps to reach end O(n) :-**
  Steps and maxreach method
  https://www.geeksforgeeks.org/minimum-number-jumps-reach-endset-2on-solution/

- **For palindrome questions consider the applying LCS.**

- **Minimum time to finish tasks without skipping two consecutive :-** Given time taken by n tasks. Find the minimum time needed to finish the tasks such that skipping of tasks is allowed, but can not skip two consecutive tasks.
  2 approaches to solve these kind of questions :-
  1) Using including excluding concept
     Let minTime(i) be minimum time to finish till i'th task. It can be written as minimum of two values.

     Minimum time if i'th task is included in list, let this time be incl(i)
     Minimum time if i'th task is excluded from result, let this time be excl(i)
     minTime(i) = min(excl(i), incl(i))
     Result is minTime(n-1) if there are n tasks and indexes start from 0.

     incl(i) can be written as below.

```
// There are two possibilities
// (a) Previous task is also included
// (b) Previous task is not included
incl(i) = min(incl(i-1), excl(i-1)) +
              arr[i] // Since this is inclusive
                     // arr[i] must be included
```

     excl(i) can be written as below.

```
// There is only one possibility (Previous task must be
// included as we can't skip consecutive tasks.
```

excl(i) = incl(i-1)
A simple solution is to make two tables incl[] and excl[] to store times for tasks. Finally return minimum of incl[n-1] and excl[n-1]. This solution requires O(n) time and O(n) space.

```
int minTime(int arr[], int n)
{
    if (n <= 0)
        return 0;
    int incl = arr[0];
    int excl = 0;
    for (int i=1; i<n; i++)
    {
       int incl_new = arr[i] + min(excl, incl);
       int excl_new = incl;
       incl = incl_new;
       excl = excl_new;
    }
    return min(incl, excl);
}
```

2) Always include an index i and return min(arr[n-1], arr[n-2])
(My solution) :-

```
dp[0] = arr[0];
dp[1] = arr[1];

for(int i = 2; i < n; i++)
    dp[i] = arr[i] + min(dp[i-1], dp[i-2]);

cout << min(dp[n-1], dp[n-2]) << endl;
```

Another example :- Minimum sum subsequence such that at least one of every four consecutive elements is picked

```
int minSum(int arr[], int n)
{
    int dp[n];
    if (n == 1)
        return arr[0];
    if (n == 2)
        return min(arr[0], arr[1]);
    if (n == 3)
        return min(arr[0], min(arr[1], arr[2]));
    if (n == 4)
        return min(min(arr[0], arr[1]),
                   min(arr[2], arr[3]));

    dp[0] = arr[0];
    dp[1] = arr[1];
    dp[2] = arr[2];
    dp[3] = arr[3];
```

```
        for (int i = 4; i < n; i++)
            dp[i] = arr[i] + min(min(dp[i - 1], dp[i - 2]),
                                 min(dp[i - 3], dp[i - 4]));
        return min(min(dp[n - 1], dp[n - 2]),
                   min(dp[n - 4], dp[n - 3]));
    }
```

- **0-1 Knapsack**

```
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }
   return K[n][W];
}
```

  - Space optimised
  - Fractional Knapsack
  - Unbounded Knapsack
    Any item can be chosen any no of times
  - Printing :-

```
          int res = K[n][W];
        printf("%d\n", res);

        w = W;
        for (i = n; i > 0 && res > 0; i--) {
            if (res == K[i - 1][w])
                continue;
            else {
                printf("%d ", wt[i - 1]);
                res = res - val[i - 1];
                w = w - wt[i - 1];
            }
        }
```

- **Distinct Subsequences O(n) | O(n) :-**

Let countSub(n) be count of subsequences of first n characters in input string. We can recursively write it as below.

countSub(n) = 2*Count(n-1) - Repetition

If current character, i.e., str[n-1] of str has

```
not appeared before, then
   Repetition = 0

Else:
   Repetition  =  Count(m)
   Here m is index of previous occurrence of
   current character. We basically remove all
   counts ending with previous occurrence of
   current character.
```

https://www.geeksforgeeks.org/count-distinct-subsequences/

- **Store count of occurence of an element using dp**
  https://www.geeksforgeeks.org/count-triplets-whose-sum-equal-perfect-cube/

```
dp[i][K]:= Number of occurance of K in A[i, i+1, i+2 … n]
for (int i = 0; i < n; ++i) {
        for (int j = 1; j <= K; ++j) {
            if (i == 0)
                dp[i][j] = (j == arr[i]);
            else
                dp[i][j] = dp[i - 1][j] + (arr[i] == j);
        }
    }
```

- **Coin change Problem :-**
  https://www.geeksforgeeks.org/coin-change-dp-7/

```
int count( int S[], int m, int n )
{
    if (n == 0)
        return 1;
    if (n < 0)
        return 0;
    if (m <=0 && n >= 1)
        return 0;
    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}
```

```
-> can be Space Optimised O(n)
```

Important :-
Here {1,2} and {2,1} are considered same. This has been implemented by first taking m items of the array and then considering for the next element.
If They were to be considered as different we can do it by making a 1D array and doing the following :-

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

```
        }
    }
```

- **Longest Common Substring :-**

```
for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                LCSuff[i][j] = 0;

            else if (X[i-1] == Y[j-1])
            {
                LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
                result = max(result, LCSuff[i][j]);
            }
            else LCSuff[i][j] = 0;
        }
    }
```

https://www.geeksforgeeks.org/longest-common-substring-dp-29/

  - Using Suffix Tree in O(m+n)

- **Egg Dropping Problem :-**
  - Recursive (n eggs, k floors)
    https://www.geeksforgeeks.org/egg-dropping-puzzle-dp-11/
    When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn't break.

    1) If the egg breaks after dropping from xth floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to x-1 floors and n-1 eggs
    2) If the egg doesn't break after dropping from the xth floor, then we only need to check for floors higher than x; so the problem reduces to k-x floors and n eggs.

    Since we need to minimize the number of trials in worst case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials

```
int eggDrop(int n, int k)
{
    if (k == 1 || k == 0)
        return k;

    if (n == 1)
        return k;

    int min = INT_MAX, x, res;
    for (x = 1; x <= k; x++)
    {
        res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));
        if (res < min)
```

```
                min = res;
            }
            return min + 1;
        }
```

- O(1) formula (2 eggs, K floors)
  Min try = x
  Then (x*(x+1))/2 >= k
-

- **In Grid Problems if we can move in all directions then consider using Graph theory instead of DP.**
  DP can be used only in memoization form and only when this condition satisfies :- if we can go from cell 1 to cell 2 then we cant go from cell 2 to cell 1.

- **Edit Distance :-** Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

  Insert
  Remove
  Replace
  All of the above operations are of equal cost.

```
int editDistDP(string str1, string str2, int m, int n)
{
    int dp[m+1][n+1];
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            if (i==0)
                dp[i][j] = j;  // Min. operations = j
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            else
                dp[i][j] = 1 + min(dp[i][j-1],  // Insert
                                   dp[i-1][j],  // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }
    return dp[m][n];
}
```

- **Find length of longest subsequence of one string which is substring of another string :-**

Let n be length of X and m be length of Y. Create a 2D array ‘dp[][]’ of m + 1 rows and n + 1 columns. Value dp[i][j] is maximum length of subsequence of X[0….j] which is substring of Y[0….i]. Now for each cell of dp[][] fill value as :

```
for (i = 1 to m)
  for (j = 1 to n)
    if (x[i-1] == y[j - 1])
      dp[i][j] = dp[i-1][j-1] + 1;
    else
      dp[i][j] = dp[i][j-1];
```

```
int maxSubsequenceSubstring(char x[], char y[],
                            int n, int m)
{
    int dp[MAX][MAX];

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

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (x[j - 1] == y[i - 1])
                dp[i][j] = 1 + dp[i - 1][j - 1];
            else
                dp[i][j] = dp[i][j - 1];
        }
    }
    int ans = 0;
    for (int i = 1; i <= m; i++)
        ans = max(ans, dp[i][n]);

    return ans;
}
```

- **WildCard Pattern Matching :-**
  https://www.geeksforgeeks.org/wildcard-pattern-matching/
  Given a text and a wildcard pattern, implement wildcard pattern matching algorithm that finds if wildcard pattern is matched with text. The matching should cover the entire text (not partial text).

  The wildcard pattern can include the characters ‘?’ and ‘*’
  ‘?’ – matches any single character
  ‘*’ – Matches any sequence of characters (including the empty sequence)

  - O(n*m) DP solution :-
    https://www.geeksforgeeks.org/wildcard-pattern-matching/
    Algorithm :-

```
// If current characters match, result is same as
// result for lengths minus one. Characters match
// in two cases:
// a) If pattern character is '?' then it matches
//    with any character of text.
// b) If current characters in both match
```

```
if ( pattern[j – 1] == ‘?’) ||
     (pattern[j – 1] == text[i - 1])
    T[i][j] = T[i-1][j-1]

// If we encounter ‘*’, two choices are possible-
// a) We ignore ‘*’ character and move to next
//    character in the pattern, i.e., ‘*’
//    indicates an empty sequence.
// b) '*' character matches with ith character in
//     input
else if (pattern[j – 1] == ‘*’)
    T[i][j] = T[i][j-1] || T[i-1][j]

else // if (pattern[j – 1] != text[i - 1])
    T[i][j]  = false

Code :-
bool strmatch(char str[], char pattern[],
              int n, int m)
{
    // empty pattern can only match with
    // empty string
    if (m == 0)
        return (n == 0);
    bool lookup[n + 1][m + 1];
    memset(lookup, false, sizeof(lookup));

    // empty pattern can match with empty string
    lookup[0][0] = true;

    // Only '*' can match with empty string
    for (int j = 1; j <= m; j++)
        if (pattern[j - 1] == '*')
            lookup[0][j] = lookup[0][j - 1];

    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            if (pattern[j - 1] == '*')
                lookup[i][j] = lookup[i][j - 1] ||
                               lookup[i - 1][j];

            else if (pattern[j - 1] == '?' ||
                    str[i - 1] == pattern[j - 1])
                lookup[i][j] = lookup[i - 1][j - 1];

            else lookup[i][j] = false;
        }
    }

    return lookup[n][m];
```

```
    }
```

- **Wildcard Pattern Matching | Linear Time and Constant Space :-**

ALGORITHM | (STEP BY STEP)

Step – (1) : Let i be the marker to point at the current character of the text.
Let j be the marker to point at the current character of the pattern.
Let index_txt be the marker to point at the character of text on which we encounter '*' in pattern.
Let index_pat be the marker to point at the position of '*' in the pattern.

NOTE : WE WILL TRAVERSE THE GIVEN STRING AND PATTERN USING A WHILE LOOP

Step – (2) : At any instant if we observe that txt[i] == pat[j], then we increment both i and j as no operation needs to be performed in this case.

Step – (3) : If we encounter pat[j] == '?', then it resembles the case mentioned in step – (2) as '?' has the property to match with any single character.

Step – (4) : If we encounter pat[j] == '*', then we update the value of index_txt and index_pat as '*' has the property to match any sequence of characters (including the empty sequence) and we will increment the value of j to compare next character of pattern with the current character of the text.(As character represented by i has not been answered yet).

Step – (5) : Now if txt[i] == pat[j], and we have encountered a '*' before, then it means that '*' included the empty sequence, else if txt[i] != pat[j], a character needs to be provided by '*' so that current character matching takes place, then i needs to be incremented as it is answered now but the character represented by j still needs to be answered, therefore, j = index_pat + 1, i = index_txt + 1 (as '*' can capture other characters as well), index_txt++ (as current character in text is matched).

Step – (6) : If step – (5) is not valid, that means txt[i] != pat[j], also we have not encountered a '*' that means it is not possible for the pattern to match the string. (return false).

Step – (7) : Check whether j reached its final value or not, then return the final answer.

```
bool strmatch(char txt[], char pat[], int n, int m)
{
    if (m == 0)
        return (n == 0);

    int i = 0, j = 0, index_txt = -1,
        index_pat = -1;

    while (i < n) {
        // For step - (2, 5)
        if (j < m && txt[i] == pat[j]) {
            i++;
            j++;
        }
```

```
        // For step - (3)
        else if (j < m && pat[j] == '?') {
            i++;
            j++;
        }
        // For step - (4)
        else if (j < m && pat[j] == '*') {
            index_txt = i;
            index_pat = j;
            j++;
        }
        // For step - (5)
        else if (index_pat != -1) {
            j = index_pat + 1;
            i = index_txt + 1;
            index_txt++;
        }
        // For step - (6)
        else {
            return false;
        }
    }
    // For step - (7)
    while (j < m && pat[j] == '*') {
        j++;
    }
    if (j == m) {
        return true;
    }
    return false;
}
```