Dynamic Programming (51 probs)

- max length chain
- Kadane's algo
- Interleaved strings
- stocks buy & sell
- minimum no of jumps
- trapping rain water
- max-product subarray
- max-rectangle
- partition equal sum subset
- #ways to reach nth stair
- job-sequencing
- egg dropping puzzle
- Consecutive #1's not allowed
- Palindromic partitioning
- special keyboard
- SCS
- Longest Common Substring
- Max profit
- Get-minimum Squares
- Total decoding Msg
- box stacking
- wildcard pattern matching
- 0-1 Knapsack
- Unique BST
- LCS
- maximize the cut segments
- coin change
- smallest window in a string containing all characters of another string
- edit distance
- #coins
- max sum Increasing Subsequence
- Max path sum in matrix
- Wordbreak
- Perfect-sum problem
- Longest bitonic Subsequence
- Mobile Numeric keyboard
- Longest Arithmetic Progression
- Min hops to reach dest.
- Boolean parenthesisation
- Strictly ↑ Array
- Jump game
- Snake & Ladder
- MCM
- form a palindrome
- Longst valid parenthesis
- find all possible palindromic partition of string
- Subset Sum problem
- Brackets in MCM
- Bellman ford
- Word wrap
- Painter's Partition problem

+ Striver Playlist ✓

Dynamic Programming
- Smart Recursion
- Invented by bellman ford
- store results of subproblems
- Eg → Linear time complexity

Activity Selection Problem

* Pre-req : Longest Increasing Subsequence

DP → $O(n^2)$ time $O(n)$ space

Greedy → $O(n \log n)$ time $O(1)$ space ✓

646. Maximum Length of Pair Chain

Medium 1971 97 Add to List Shar

You are given an array of `n` pairs `pairs` where `pairs[i] = [left_i, right_i]` and `left_i < right_i`.

A pair `p2 = [c, d]` **follows** a pair `p1 = [a, b]` if `b < c`. A **chain** of pairs can be formed in this fashion.

Return *the length longest chain which can be formed.*

You do not need to use up all the given intervals. You can select pairs in any order.

Example 1:

curr ↓ ↓ curr ↑ (2)

```
Input: pairs = [[1,2],[2,3],[3,4]]
Output: 2
Explanation: The longest chain is [1,2] -> [3,4].
```

Example 2:

```
Input: pairs = [[1,2],[7,8],[4,5]]
Output: 3
Explanation: The longest chain is [1,2] -> [4,5] -> [7,8].
```

```
bool cmp(vector<int> a, vector<int> b)
{
   return a[1] < b[1];
}

class Solution
{
    public:
    int findLongestChain(vector<vector<int>>& pairs)
    {
        int n = pairs.size();
        sort(pairs.begin(), pairs.end(), cmp); //sort the list

        //initialization
        int count = 1;
        int cur = pairs[0][1];

        for(int i=1; i<n; i++) {
            if(cur < pairs[i][0]) { //b<c
                cur = pairs[i][1];
                count++;
            }
        }
        return count;
    }
};
```

we are sorting the pairs based on finish time & then

Initialise count as 1

cur = first pair first index ie first pair start time as i start from 1. we check if start of index 0 < start of index i Increment the curr & update count..

```
//Using Greedy Approach
bool compare(struct val a,struct val b){
    return a.second<b.second;
}
int maxChainLen(struct val p[],int n)
{
    sort(p,p+n,compare);
    int l=1,prev=p[0].second;
    for(int i=1;i<n;i++)
        if(p[i].first>prev)
        {
            prev=p[i].second;
            l++;
        }
    return l;
}
```

```
bool comparator(struct val a, struct val b){
    return a.first < b.first;
    // or return a.second < b.second;
}
int maxChainLen(struct val arr[],int n)
{
    sort(arr,arr+n,comparator);
    int t[n];
    t[0] = 1;
    int ans = 1;
    for(int i=1;i<n;i++){<br>
        t[i] = 1;
        for(int j=0;j<i;j++){<br>
            if(arr[j].second < arr[i].first){
                t[i] = max(t[i] , t[j] + 1);
                ans = max(ans , t[i]);
            }
        }
    }
    return ans;
}
```

Kadane's Algorithm O(n)

53. Maximum Subarray

Easy 17310 817 Add to List Share

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Example 2:

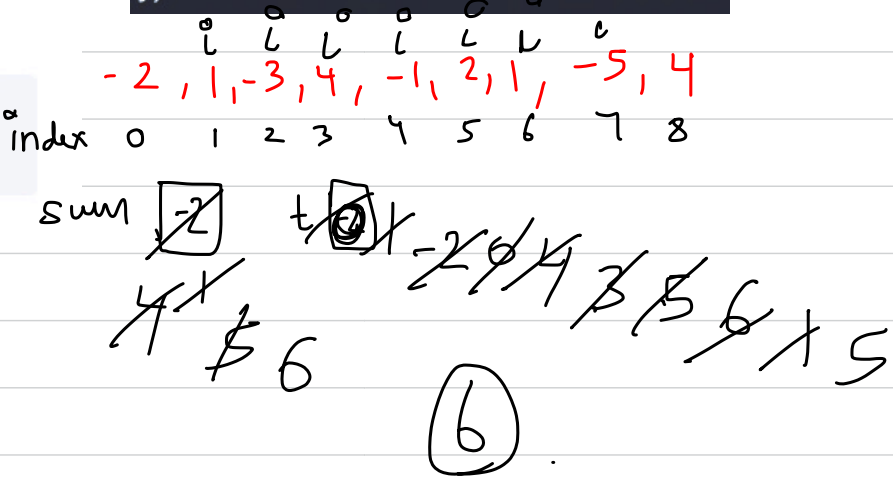
```
Input: nums = [1]
Output: 1
```

Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
```

```cpp
class Solution
{
    public:
    int maxSubArray(vector<int> &nums)
    {
        int sum = nums[0], t = nums[0];
        for (int i = 1; i < nums.size(); i++)
        {
            t = max(t + nums[i], nums[i]);
            sum = max(sum, t);
        }
        return sum;
    }
};
```

```cpp
class Solution{
    public:
    long long maxSubarraySum(int arr[], int n){
        int sum=arr[0],t=arr[0];
        for(int i=1;i<n;i++){
            if(t<0) t=0;
            t+=arr[i];
            sum = max(sum,t);
        }
        return sum;
    }
};
```



978. Longest Turbulent Subarray

Medium 1160 160 Add to List Sha

Given an integer array `arr`, return *the length of a maximum size turbulent subarray of* `arr`.

A subarray is **turbulent** if the comparison sign flips between each adjacent pair of elements in the subarray.

More formally, a subarray `[arr[i], arr[i + 1], ..., arr[j]]` of `arr` is said to be turbulent if and only if:

- For `i <= k < j`:
  - `arr[k] > arr[k + 1]` when `k` is odd, and
  - `arr[k] < arr[k + 1]` when `k` is even.
- Or, for `i <= k < j`:
  - `arr[k] > arr[k + 1]` when `k` is even, and
  - `arr[k] < arr[k + 1]` when `k` is odd.

```
class Solution {
public:
    int maxTurbulenceSize(vector<int>& v)
    {
        int n=v.size();
        if(n<=1) return n;
        vector<int>dp(n);
        dp[0]=1;
        if(v[1]==v[0]) //if their is no mountain/valley
            dp[1]=1; //keep it same
        else
            dp[1]=2; //else update the answer to 2 and store in dp
        int ans=max(dp[0],dp[1]);//ans = max of both
        //run a for loop from index 2
        for(int i=2;i<n;i++)
        {
            bool peak=v[i-1]<v[i] and v[i-1]<v[i-2]; //peak
            bool mountain=v[i-1]>v[i] and v[i-1]>v[i-2];//mountain

            if(peak or mountain)
                dp[i]=dp[i-1]+1;
            else //neighbours neither totally peak nor mountain
                if(v[i]!=v[i-1]) //if they are not same
                    dp[i]=2;
                else
                    dp[i]=1; //if they are same
            ans=max(ans,dp[i]);
        }
        return ans;
    }
};
```

97. Interleaving String

Medium 3425 182 Add to List Sha

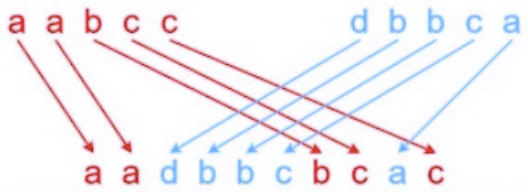
Given strings `s1`, `s2`, and `s3`, find whether `s3` is formed by an **interleaving** of `s1` and `s2`.

An **interleaving** of two strings `s` and `t` is a configuration where they are divided into **non-empty** substrings such that:

- $s = s_1 + s_2 + ... + s_n$
- $t = t_1 + t_2 + ... + t_m$
- $|n - m| <= 1$
- The **interleaving** is $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + ...$ or $t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + ...$

Note: `a + b` is the concatenation of strings `a` and `b`.

Example 1:



Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
Output: true

bfs version faster than dfs

```
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3)
    {
        int l1 = s1.size(), l2 = s2.size(), l3 = s3.size();
        if (l1 + l2 != l3) return false;
        vector<vector<bool>> visited(l1 + 1, vector<bool>(l2 + 1, false));
        queue<pair<int, int>> q;
        q.push(pair<int, int>(0, 0));
        while(!q.empty()){
            auto p = q.front();q.pop();
            if (p.first == l1 and p.second == l2)return true;
            if (visited[p.first][p.second]) continue;
            if (p.first < l1 && s1[p.first] == s3[p.first + p.second])
                q.push(pair<int, int>(p.first + 1, p.second));
            if (p.second < l2 && s2[p.second] == s3[p.first + p.second])
                q.push(pair<int, int>(p.first, p.second + 1));
            visited[p.first][p.second] = true;
        }
        return false;
    }
};
```

dp approach →

```
bool isInterleave(string s1, string s2, string s3) {

    if(s3.length() != s1.length() + s2.length())
        return false;

    bool table[s1.length()+1][s2.length()+1];

    for(int i=0; i<s1.length()+1; i++)
        for(int j=0; j< s2.length()+1; j++){
            if(i==0 && j==0)
                table[i][j] = true;
            else if(i == 0)
                table[i][j] = ( table[i][j-1] && s2[j-1] == s3[i+j-1]);
            else if(j == 0)
                table[i][j] = ( table[i-1][j] && s1[i-1] == s3[i+j-1]);
            else
                table[i][j] = (table[i-1][j] && s1[i-1] == s3[i+j-1] ) || (table[i][j-1] && s2[j-1] == s3[i+j-1] );
        }

    return table[s1.length()][s2.length()];
}
```

MAX PRODUCT SUBARRAY

```cpp
class Solution {
public:
    // TIME COMPLEXITY:- O(N)
    // SPACE COMPLEXIY:- O(1)
    int maxProduct(vector<int>& nums) {
        int ans = nums[0],max_prod = nums[0],min_prod = nums[0]; // initialize max product,min product and answer
        for(int i=1;i<nums.size();i++){
            if(nums[i]<0) // if number is negative, we will swap max prod and min prod
                swap(max_prod,min_prod);
            max_prod = max(nums[i],max_prod*nums[i]); // find current max prod each time
            min_prod = min(nums[i],min_prod*nums[i]); // find current min prod each time
            ans = max(ans,max_prod); // store the maximum product each time
        }
        return ans;
    }
};
```

Best time to buy and sell stock DP

```cpp
class Solution{
     public:
    int maxProfit(vector<int> prices)
    {
        int maxCur = 0, maxSoFar = 0;
        for(int i = 1; i < prices.size(); i++)
        {
            maxCur = max(0, maxCur += prices[i] - prices[i-1]);
            maxSoFar = max(maxCur, maxSoFar);
        }
        return maxSoFar;
    }
};
```

```cpp
#define vvi vector<vector<int>>
#define vi vector<int>
class Solution
{
    public:
    vvi stockBuySell(vi arr, int n)
    {
        vvi result;
        vi v;
        for (int i = 1; i < n; i++)
        {
            if (arr[i] > arr[i-1])
            {
                v.push_back(i-1);
                v.push_back(i);
                result.push_back(v);
            }
            v.clear();
        }
        return result;
    }
};
```

Here we are initialising the current max as 0 and max so far as 0. We will iterate the vector prices from left to right starting from index 1. So we will update Current max as max of 0 and maxCur+ current - prev price.
MaxSoFar will get updated as max of current max and max so far.
At the end we will return the maxSoFar.

Minimum Number of Jumps to reach end O(n) time solution

The minimum number of jumps needed to reach the starting index is 0 so if n<=1 return 0. If the first index of the array is 0 then return -1 as we don't even have enough fuel to go ahead.
We store the all time maximum reachable index in the array in the integer variable known as maxReach.
We store the number of steps we can still take in the integer variable called step.
Jump stores the number of jumps necessary to reach the maximal reachable position.
We start traversing the array from index 1 to the end of the array. If we have reached the end of the array that is if I==n-1 we return jump. maxReach is updated as max(maxReach, I+arr[I])

```
class Solution{
    public:
      int minJumps(int arr[], int n)
      {
        if(n==0||n==1) return 0;
        if(arr[0]==0) return -1;
       int maxReach=arr[0];
       int steps=arr[0];
       int jumps=0;

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

           if(i==n-1) return jumps+1;
           steps--;
           maxReach=max(maxReach,arr[i]+i);
           if(steps==0){
               if(i==maxReach) return -1;
               jumps++;
               steps=maxReach-i;
           }
       }
    }
};
```

Here we are checking that if n==0 or n==1 then we are returning 0
If the first element is 0 then we have to return 0;

We decrement the steps
Update the max reach as max of max reach and arr[l]+i

check if there are no more steps that we can take, then check if the max reach becomes equal to I we have to return -1
Simply increment jump as step are 0. And update the step as max reach-i;

```
class Solution {
public:
    int jump(vector<int>& nums)
    {

        if(nums.size()<2) return 0;   //base case

        //initialize jump=1 , we are taking jump from 0th index to the range mxjump
        //currjump, we can take jump from particular  index
        //mxjump , we cango up to maximum
        // jump to count no. of jump
        int jump=1,n=nums.size(),currjmp=nums[0],mxjmp=nums[0];

        int i=0;

        //till we reach last index, NOTE: Not necessary to cross last index
        while(i<n-1)
        {
            mxjmp=max(mxjmp,i+nums[i]);

            if(currjmp==i) //we have to take jump now because our currjump now ends.
            {
                jump++;//increment in jump
                currjmp=mxjmp; //assign new maxjmp to currjmp
            }
            i++;
        }
        return jump;
    }
};
```

1029. Two City Scheduling

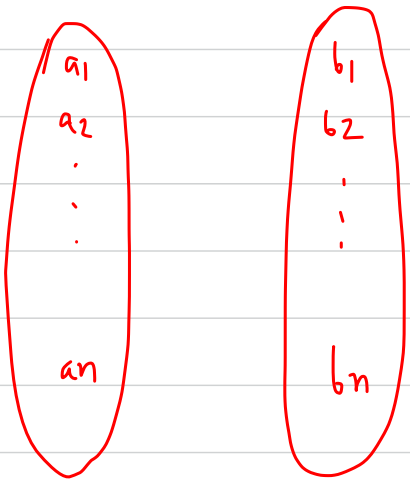
Medium 2274 225 Add to List Sha

A company is planning to interview `2n` people. Given the array `costs` where `costs[i] = [aCost_i, bCost_i]`, the cost of flying the `i^th` person to city `a` is `aCost_i`, and the cost of flying the `i^th` person to city `b` is `bCost_i`.

Return *the minimum cost to fly every person to a city* such that exactly `n` people arrive in each city.

```cpp
class Solution {
public:
  //Bottom-Up dp
    int twoCitySchedCost(vector<vector<int>>& costs){
        int n = costs.size()/2;
        vector<vector<int>>dp(n+1,vector<int>(n+1,0));
        for (int i = 1; i <= n; i++)dp[i][0] = dp[i - 1][0] + costs[i - 1][0];
        for (int j = 1; j <= n; j++)dp[0][j] = dp[0][j - 1] + costs[j - 1][1];

        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                dp[i][j] = min(dp[i-1][j]+costs[i+j-1][0], dp[i][j-1] + costs[i+j-1][1]);

        return dp[n][n];

    }
};
```

This is a 3D dp question inspired from cherry pickup II in this we are given 2n people and we are given a cost array where each element of the array is a pair A COST and B COST where A COST means cost of flying the iTH person to city a is A cost and cost of flying the iTh person in city B is B COST.
We have to return the minimum cost to fly every person to a city such that exactly n people arrive in each city.



as we know costs vector is a 2D vector & it's size is 2n

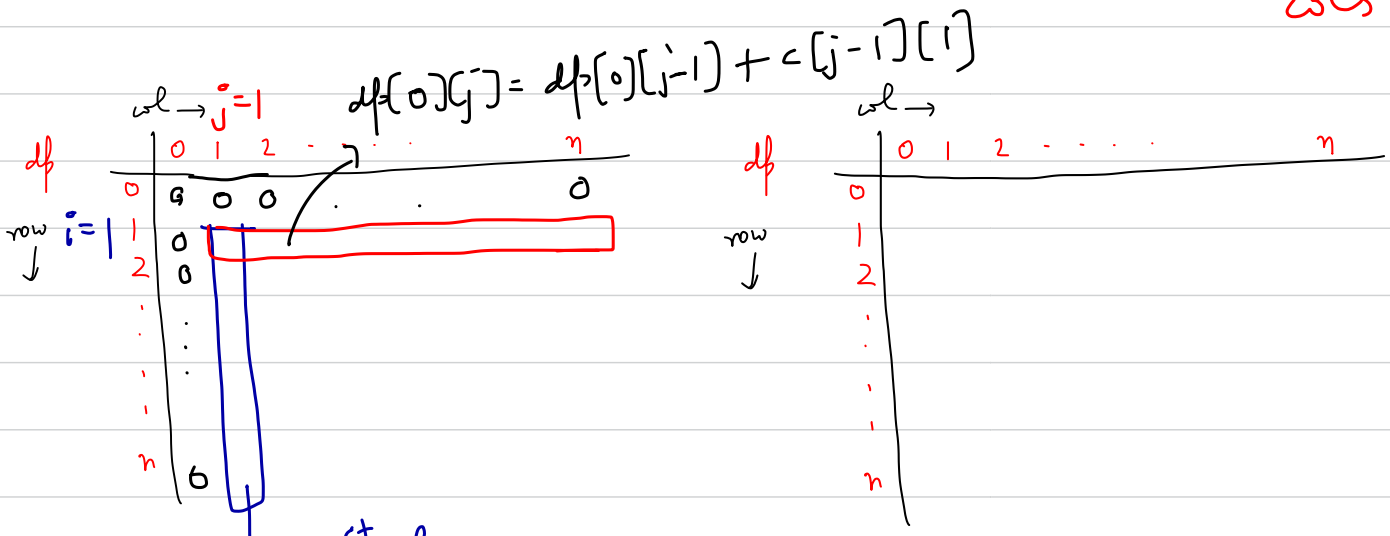
so let n = costs.size() / 2 ;

we will make a dp vector

vvi dp(n+1, vi(n+1, 0)) → each value 0.

n+1 rows

n+1 cols



we will fill 1st col.

dp[i][0] = dp[i-1][0] + c[i-1][0],

How much money can we save if we fly a person to A vs. B? To minimize the total cost, we should fly the person with the maximum saving to A, and with the minimum - to B.

Example: [30, 100], [40, 90], [50, 50], [70, 50].
Savings: 70, 50, 0, -20.

Obviously, first person should fly to A, and the last - to B.

Solution

We sort the array by the difference between costs for A and B. Then, we fly first `N` people to A, and the rest - to B.

```
int twoCitySchedCost(vector<vector<int>>& cs, int res = 0) {
  sort(begin(cs), end(cs), [](vector<int> &v1, vector<int> &v2) {
    return (v1[0] - v1[1] < v2[0] - v2[1]);
  });
  for (auto i = 0; i < cs.size() / 2; ++i) {
    res += cs[i][0] + cs[i + cs.size() / 2][1];
  }
  return res;
}
```

Optimized Solution

Actually, we do not need to perfectly sort all cost differences, we just need the biggest savings (to fly to A) to be in the first half of the array. So, we can use the quick select algorithm (`nth_element` in C++) and use the middle of the array as a pivot.

This brings the runtime down from 8 ms to 4 ms (thanks @popeye1 for the tip!)

```
int twoCitySchedCost(vector<vector<int>>& cs, int res = 0) {
  nth_element(begin(cs), begin(cs) + cs.size() / 2, end(cs), [](vector<int> &a, vector<int> &b) {
    return (a[0] - a[1] < b[0] - b[1]);
  });
  for (auto i = 0; i < cs.size() / 2; ++i) {
    res += cs[i][0] + cs[i + cs.size() / 2][1];
  }
  return res;
}
```

Complexity Analysis

Runtime: *O(n log n)*. We sort the array then go through it once. The second solution has a better average case runtime.
Memory: *O(1)*. We sort the array in-place.

O(nlogn) time and O(1) space the most optimised using greedy approach

It is not necessary that DP will give the best answer, sometimes the greedy approach outperforms every other approach. But this certainly does not mean that DP solution is underrated.

We are using the nth_element method of vector and passing the parameters start index, middle index and end index

```
class Solution
{
    public:
    static bool comp(vector<int>&a, vector<int>&b)
    {
        return (a[0] - a[1] < b[0] - b[1]);
    }
    int twoCitySchedCost(vector<vector<int>>& cs, int res = 0)
    {
        nth_element(begin(cs), begin(cs)+cs.size()/2, end(cs), comp);
        for (auto i=0; i < cs.size()/2; ++i)
            res += cs[i][0] + cs[i + cs.size()/2][1];
        return res;
    }
};
```

This is my approach where I have used a comparator Class which returns boolean output.
It compares that if cost of ITH PERSON IN CITY A IS LESS THAN COST OF JTH PERSON IN CITY A AND COST OF ITH PERSON IN CITY B IS LESS THAN COST OF JTH PERSON IN CITY B THEN ONLY it returns true, else it will return false.

Using quick select algorithm, we will take the elements before the pivot and take their first index and add it to element after pivot and their second index
at last return the result.

Trapping Rainwater

42. Trapping Rain Water

Hard 15922 227 Add to List S

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

```
class Solution {
public:
    int trap(vector<int>& height)
    {
        int n = height.size(),ans=0;
        vector<int> left (n), right (n);
        left[0] = height[0];
        for (int i=1; i<n;i++) left[i] = max ( left[i-1], height[i] );
        right[n-1] = height[n-1];
        for (int i=n-2;i>=0;i--) right[i] = max(right[i+1],height[i]);
        for ( int i = 0; i < n; i++ ) ans += min( left[i], right[i] ) - height[i];
        return ans;
    }
};
```

What we are doing here is we are storing the max till now at each index in an array and call it as left as we are traversing from Left to right

We are storing the max till now from the right and moving towards left as we progress and we call this array as right.

Now what we want to do is keep traversing from left to right from 0 to n-1 and then update the answer with min of these two array indices and then subtracting the height[I] as that is what we want na

This will result into the max water that can be trapped.

Maximum Rectangle

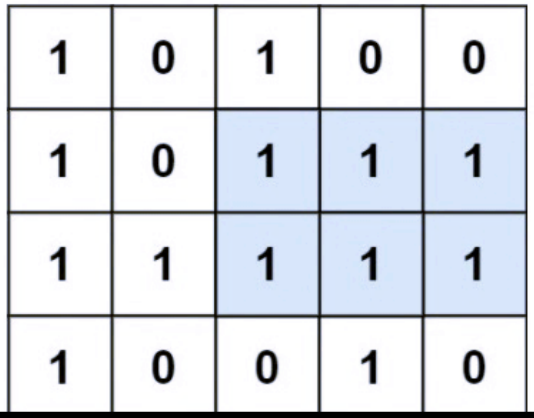
Maximum area rectangle in a histogram is the prerequisite for this question

85. Maximal Rectangle

Hard 5929 101 Add to List Shar

Given a `rows x cols` binary `matrix` filled with `0`'s and `1`'s, find the largest rectangle containing only `1`'s and return *its area*.

Example 1:




```
class Solution {
public:
    int largestRectangleArea(vector<int>& v)
    {
        int n = v.size(), area=0, h, l;
        stack<int> s;
        for (int i = 0; i <= n; i++)
        {
            while(i==n or (!s.empty() and v[s.top()] > v[i]) )
            {
                if (s.empty() and i==n) h=0, i++;
                else h = v[s.top()], s.pop();
                l = s.empty() ? -1 : s.top();
                area = max(area, h * (i - l - 1));
            }
            s.push(i);
        }
        return area;
    }
};
```

5 3 7 1 4

What we do in maximum area histogram is we are given with the vector containing all the values which represent the height of the towers.
I have used variable n as the size of the vector v. I have initialised area variable as 0 as we don't know what is area as of now. Also we have taken two variables h and l. H represents high and l represents low.
We have taken a stack of integer type and named as s.

We will write a for loop which will start from 1 to the end of the vector length. And inside this for loop we will run a while loop which will only run if we are at the end of the vector or when the stack is not empty and v[stacktop]>v[I]
In the while loop we will check if the stack is empty that is if we are at the end of the vector and also will check the same that are we at I=n then in that case, we will increment I and set the high as 0.
else we will set the high as vector indexed at top of stack. And we will pop off the top from the stack.
If the stack is empty we will set the low as -1 otherwise low will be the top of the stack. Area will be the maximum of area and high times I-low-1. And at the end we will push I onto the stack. And return the area.

1691. Maximum Height by Stacking Cuboids

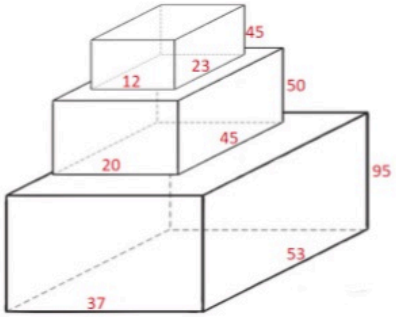
Hard 405 14 Add to List Share

Given `n` `cuboids` where the dimensions of the `i`th cuboid is `cuboids[i] = [width_i, length_i, height_i]` (**0-indexed**). Choose a **subset** of `cuboids` and place them on each other.

You can place cuboid `i` on cuboid `j` if `width_i <= width_j` and `length_i <= length_j` and `height_i <= height_j`. You can rearrange any cuboid's dimensions by rotating it to put it on another cuboid.

Return *the **maximum height** of the stacked* `cuboids`.

Example 1:




```
Input: cuboids = [[50,45,20],[95,37,53],[45,23,12]]
Output: 190
Explanation:
Cuboid 1 is placed on the bottom with the 53x37 side facing down with
height 95.
```

```cpp
class Solution
{
    public:
    int maxHeight(vector<vector<int>> &C)
    {
        int n = C.size();
        for (auto &c : C) sort(c.begin(), c.end());
        sort(C.begin(), C.end());
        vector<int> dp(n, 0);
        for (int i = 0; i < n; i++)
        {
            dp[i] = C[i][2];
            for (int j = 0; j <= i - 1; j++)
                if ((C[i][0] >= C[j][0]) and (C[i][1] >= C[j][1]) and (C[i][2] >= C[j][2]))
                    dp[i] = max(dp[i], dp[j] + cuboids[i][2]);
        }
        int ans = *max_element(dp.begin(), dp.end());
        return ans;
    }
};
```

```
int lis(int arr[], int n)
{
   vector<int>seq;
   seq.push_back(arr[0]);

   for(int i = 1;i<n;i++)
   {
      if(seq.back() < arr[i])
         seq.push_back(arr[i]);

      else
      {
         int ind = lower_bound(seq.begin(), seq.end(), arr[i]) - seq.begin();
         seq[ind] = arr[i];
      }
   }

   return seq.size();
}
```

This is not a DP approach and it takes O(n log n) time as it used binary search and there are n number of elements so for each element you are doing a binary search.

We are pushing the array first element into the vector and then from the second index of the array till the end of the array we will traverse the array, and we will check if the last element of sequence is less than the present element at index I then we are ok as it is in increasing order so we will push arr[I] onto the vector. Otherwise what we will do is we will find the lower_bound of the element at arr[I] and we will store it in index ind. and we will change the element of vector at index ind to be arr[I] and then at the end, we will return the size of the vector.

This will give the LIS of the array.

Lower bound used binary search.

```
class Solution
{
    public:
    //Function to find length of longest increasing subsequence.
    int longestSubsequence(int n, int arr[])
    {
        vector<int> v;
        v.push_back(arr[0]);
        for(int i=0;i<n;i++)
            if(arr[i]>v.back())v.push_back(arr[i]);
            else v[lower_bound(v.begin(),v.end(),arr[i])-v.begin()]=arr[i];
        return v.size();
    }
};
```

44. Wildcard Matching

Hard 4066 180 Add to List Share

Given an input string (`s`) and a pattern (`p`), implement wildcard pattern matching with support for `'?'` and `'*'` where:

- `'?'` Matches any single character.
- `'*'` Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input: s = "aa", p = "*"
Output: true
Explanation: '*' matches any sequence.
```

```cpp
class Solution {
public:
    bool isMatch(string s, string p){
        int m = s.length(), n = p.length();
        vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
        dp[0][0] = true;
        for(int i = 0; i < n; i++)dp[0][i+1] = dp[0][i] and p[i] == '*';
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(p[j] == '*') dp[i+1][j+1] = dp[i+1][j] or dp[i][j+1];
                else dp[i+1][j+1] = dp[i][j] and (s[i] == p[j] or p[j] == '?');


        return dp.back().back();
    }
};
```

```cpp
// greedy solution with idea of DFS
// starj stores the position of last * in p
// last_match stores the position of the previous matched char in s after a *
// e.g.
// s: a c d s c d
// p: * c d
// after the first match of the *, starj = 0 and last_match = 1
// when we come to i = 3 and j = 3, we know that the previous match of * is actually wrong,
// (the first branch of DFS we take is wrong)
// then it resets j = starj + 1
// since we already know i = last_match will give us the wrong answer
// so this time i = last_match+1 and we try to find a longer match of *
// then after another match we have starj = 0 and last_match = 4, which is the right solution
// since we don't know where the right match for * ends, we need to take a guess (one branch in DFS),
// and store the information(starj and last_match) so we can always backup to the last correct place and take another guess
bool isMatch(string s, string p) {
        int i = 0, j = 0;
        int m = s.length(), n = p.length();
        int last_match = -1, starj = -1;
        while (i < m){
            if (j < n && (s[i] == p[j] || p[j] == '?')){
                i++; j++;
            }
            else if (j < n && p[j] == '*'){
                starj = j;
                j++;
                last_match = i;
            }
            else if (starj != -1){
                j = starj + 1;
                last_match++;
                i = last_match;
            }
            else return false;
        }
        while (p[j] == '*' && j <n) j++;
        return j == n;
    }
```

```cpp
vector<vector<int> > dp;
int finding(string& s, string& p, int n, int m)
{
    // return 1 if n and m are negative
    if (n < 0 && m < 0)
        return 1;

    // return 0 if m is negative
    if (m < 0)
        return 0;

    // return n if n is negative
    if (n < 0)
    {
        // while m is positive
        while (m >= 0)
        {
            if (p[m] != '*')
                return 0;
            m--;
        }
        return 1;
    }
```

DP Memoization solution on your left

1143. Longest Common Subsequence

Medium 5247 60 Add to List Share

Given two strings `text1` and `text2`, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return `0`.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, `"ace"` is a subsequence of `"abcde"`.

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

By Top-Down :

```
vector<vector<int>> dp;
int lcs(int i,int j,string &a, string &b)
{
    if(i==-1 || j==-1)    return 0;
    if(dp[i][j] != -1)  return dp[i][j];

    if(a[i] == b[j])    return dp[i][j] = 1+lcs(i-1,j-1,a,b);
    return dp[i][j] = max(lcs(i-1,j,a,b), lcs(i,j-1,a,b));
}
int longestCommonSubsequence(string text1, string text2) {
    int n1 = text1.size();
    int n2 = text2.size();
    dp.resize(n1,vector<int>(n2,-1));

    return lcs(n1-1,n2-1,text1,text2);
```

By Bottom-Up :

```
    int n1 = text1.size();
    int n2 = text2.size();
    vector<vector<int>> dp(n1+1,vector<int>(n2+1,0));


    for(int i=1;i<=n1;i++)
    {
        for(int j=1;j<=n2;j++)
        {
            if(text1[i-1] == text2[j-1])
                dp[i][j] = 1+dp[i-1][j-1];
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);

        }
    }
    return dp[n1][n2];
```

Space optimised DP Time: O(mn) space: O(max(m,n))

```
class Solution
{
    public:
    vector<vector<int>> dp;
    int longestCommonSubsequence(string x, string y)
    {
        int m = x.size();
        int n = y.size();
        vector<vector<int>> dp(2, vector<int>(n+1,0));
        bool b;


        for(int i=1;i<=m;i++){
            b=1&i;
            for(int j=1;j<=n;j++){
                if(!i or !j) dp[i][j]=0;
                else if(x[i-1]==y[j-1]) dp[b][j] = dp[1-b][j-1]+1;
                else dp[b][j] = max(dp[1-b][j], dp[b][j-1]);
            }
        }
        return dp[b][n];
    }
};
```

1092. Shortest Common Supersequence

Hard 1795 35 Add to List Share

Given two strings `str1` and `str2`, return *the shortest string that has both* `str1` *and* `str2` *as* ***subsequences***. If there are multiple valid strings, return **any** of them.

A string `s` is a **subsequence** of string `t` if deleting some number of characters from `t` (possibly `0`) results in the string `s`.

Example 1:

```
Input: str1 = "abac", str2 = "cab"
Output: "cabac"
Explanation:
str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".
str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".
The answer provided is the shortest such string that satisfies these properties.
```

Example 2:

```
Input: str1 = "aaaaaaaa", str2 = "aaaaaaaa"
Output: "aaaaaaaa"
```

c a b _ _
_ a b a c
c a b a c

a a
a a a a
a a a a

```cpp
#define vvs vector<vector<string>>
#define vs vector<string>
class Solution{
    public:
    string shortestCommonSupersequence(string& x, string& y){
        int i=0,j=0;
        string s="";
        for (char c:lcs(x,y)){
            while (x[i]!=c) s+=x[i++];
            while (y[j]!=c) s+=y[j++];
            s+=c,i++,j++;
        }
        return s+x.substr(i)+y.substr(j);
    }
    string lcs(string& x, string& y) {
        int n = x.size(), m = y.size();
        vvs dp(n+1,vs(m+1, ""));
        for (int i=0;i<n;++i)
            for (int j=0; j<m;++j)
                if (x[i]==y[j]) dp[i+1][j+1] = dp[i][j]+x[i];
                else dp[i+1][j+1]=dp[i+1][j].size()>dp[i][j+1].size()? dp[i+1][j] : dp[i][j+1];
        return dp[n][m];
    }
};
```

1. We will find the longest common subsequence of both the strings x and y using the standard DP approach. Then we declare a vector of vector of string vvs dp who is going to store our LCS.
2. we will traverse the entire memo once and update each entry one by one, each entry will contain the LCS upto that part in a string.
3. We will check if both the characters in the two strings are similar in that case, we will add what is stored previously in the dp[i][j] with x[i] i.e. the character at that particular location in dp[i+1][j+1].
4. if they are not equal in that case we will check the size of the left cell of dp[i+1][j+1] and up cell of dp[i+1][j+1] and check who has bigger string present, we copy the bigger string in the cell and then move on.
5. At the end we return dp[n][m].
6. This will return the LCS as the return type of the function is string.
7. Now we look at our function SCS, that will start with empty string s.
8. We will traverse over lcs(x,y) with character c and check inside with a while loop and keep on adding the non similar character in x then we wil add it to s and incremenet i++, likewise we will do for string y and update the pointer and add them into the string if they do not match.
9. At the end the string s contains the non matching characters which are not a part of LCS
10. Now at the end we will keep on adding LCS character C and increment both the pointers

At the end return s+ remaining letters in x and remaining letters in y.

GFG HARD

Print all LCS sequences

Hard Accuracy: 63.52% Submissions: 3667 Points: 8

You are given two strings **s** and **t**. Now your task is to print all longest common sub-sequences in lexicographical order.

Example 1:

```
Input: s = abaaa, t = baabaca
Output: aaaa abaa baaa
```

Example 2:

```
Input: s = aaa, t = a
Output: a
```

Your Task:
You do not need to read or print anything. Your task is to complete the function **all_longest_common_subsequences()** which takes string a and b as first and second parameter respectively and returns a list of strings which contains all possible longest common subsequences in lexicographical order.

Expected Time Complexity: $O(n^4)$
Expected Space Complexity: $O(K * n)$ where K is a constant less than n.

Constraints:

```
vector<string> all_longest_common_subsequences(string x, string y){
    STRING.clear();
    int m=x.length();
    int n=y.length();
    for(int i=0;i<m+1;i++)
        for(int j=0;j<n+1;j++)
            if(!i or !j) dp[i][j]=0;
    for(int i=1;i<m+1;i++)
        for(int j=1;j<n+1;j++)
            if(x[i-1]!=y[j-1]) dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
            else dp[i][j]=dp[i-1][j-1]+1;
    vector<string> result;
    string S;
    int i=m;
    int j=n;
    printLCS(result,x,y,i,j,S);
    sort(result.begin(),result.end());
    return result;
```

```
class Solution
{
    public:
    int dp[51][51];
    unordered_set<string> MAP, STRING;
    void printLCS(vector<string> &result,string x,string y,int i,int j,string S){
        if(!i or !j){
            reverse(S.begin(),S.end());
            if(STRING.find(S)!=STRING.end()) return;
            STRING.insert(S);
            result.push_back(S);
            return;
        }
        string key;
        key+=to_string(i);
        key+=to_string(j);
        key+=S;
        if(MAP.find(key)!=MAP.end()) return;
        MAP.insert(key);
        if(x[i-1]==y[j-1]) printLCS(result,x,y,i-1,j-1,S+x[i-1]);
        else if(dp[i-1][j]>dp[i][j-1]) printLCS(result,x,y,i-1,j,S);
        else if(dp[i-1][j]<dp[i][j-1])printLCS(result,x,y,i,j-1,S);
        else{
             printLCS(result,x,y,i-1,j,S);
             printLCS(result,x,y,i,j-1,S);
        }
        return;
    }
```

We have taken two unordered set MAP and STRING and we are given two Strings x and y and we need to return all the possible Longest common subsequences as one big string. We will store length of string x into m and Length of string y into n. Then we will Use a DP memo of m*n size and we will check if either or both of the I or j is 0 then we will write dp[I][j]=0 and This was the initialisation of the memo, after that from second row and

Second column we will again traverse the memo DP and if the previous diagonally upward string alphabet is not same as other string alphabet then in that case we will Take the maximum of both the left and up cell and update the current cell, otherwise dp[I][j] = 1+dp[i-1][j-1]. We will have a vector of string and we name it as result and we will have a string S. We will have two variables I and j and both will point to the length of strings X and Y respectively so I=m and j=n.
We will then call the PRINTLCS function and the parameters to that function will be the Vector of string **result**, both the strings **x** and string **y** their lengths and string **S**. The print LCS will do one thing it will take your string and firstly if any of the string x and y is NULL it will check and reverse your string S and still if the unordered set STRING contains the string it will return if not it will insert the string S into the unordered set STRING and also push this string in the vector result. At the end it will return. Now we will take string key and will will convert the length of the string into string and append to key and same we will do with length of other string and append that too into the string key. At the end we will check if the map contains that string key then we return, if not, we will add this key to the unordered set MAP of string.
Now check if the left upper diagonal box is having same string character, then intuit case, recessively call printLCS on the same string x and y but this time reduce the length of both the strings and pass I-1 and j-1 and pass the string S+x[I-1] which means that you have considered this string .
otherwise if he UP box is greater than the left box then in that case just recursively call the printLCS on same parameters just decrement the length of string x.
Else if the left box is greater than the UP box then in that case just recursively call the printLCS on same parameters just decrement the length of string y.
In the else can, just call the both cases of print LCS where once you decrement the string x length and in the other case you decrement the string y length.and finally return. As the return type of the function is VOID.

Now in the main function, sort the result from begin to end as we need in ascending order and return the result vector of strings.

Backtracking solution for leetcode subsets II

```cpp
#define vvi vector<vector<int>>
#define vi vector<int>
class Solution
{
    public:
    vvi subsetsWithDup(vi &nums)
    {
        sort(nums.begin(), nums.end());
        vvi res;
        vi v;
        subsetsWithDup(res, nums, v, 0);
        return res;
    }
    private:
    void subsetsWithDup(vvi &res, vi &nums, vi &v, int begin)
    {
        res.push_back(v);
        for (int i = begin; i != nums.size(); ++i)
            if (i == begin || nums[i] != nums[i - 1]) {
                v.push_back(nums[i]);
                subsetsWithDup(res, nums, v, i + 1);
                v.pop_back();
            }
    }
};
```

1982. Find Array Given Subset Sums

Hard 162 17 Add to List

You are given an integer `n` representing the length of an unknown array that you are trying to recover. You are also given an array `sums` containing the values of all 2^n **subset sums** of the unknown array (in no particular order).

Return *the array* `ans` *of length* `n` *representing the unknown array. If* ***multiple*** *answers exist, return* ***any*** *of them.*

An array `sub` is a **subset** of an array `arr` if `sub` can be obtained from `arr` by deleting some (possibly zero or all) elements of `arr`. The sum of the elements in `sub` is one possible **subset sum** of `arr`. The sum of an empty array is considered to be `0`.

Note: Test cases are generated such that there will **always** be at least one correct answer.

Intuition: If all the numbers in the unknown array are non-negative, it's easy to solve because `A[0]` must be `0` (empty set) and `A[1]` must be a number in the unknown array.

Hint:

- **Step 1**: Solve the problem knowning that the unknown array only has non-negative numbers.
- **Step 2**: Find a subset in the answer array whose sum is the minimal value in `A`, and turn all the numbers in this subset negative

Algorithm:

Let `mn` be the minimal number in `A`. We offset all the numbers in `A` by `-mn`, making all the numbers in `A` non-negative.

Step 1:

We store all the numbers in a `multiset<int> s`.

Now we can repeat the following process to get the unknown array:

1. Take the 2nd smallest element (say `num`) as a number in the answer `ans`.
2. Keep moving the first element (say `first`) from `s` to a new `multiset<int> tmp`, and removing `first + num` from `s`, until `s` becomes empty.
3. Now `tmp` contains all the subset sums that are formed with the rest of the unknown numbers. This `tmp` becomes the new `s`.
4. We repeat this process `n` times to fill `n` numbers into `ans`.

Now we get an array `ans` corresponding to the unknown array of the original `A` with offset `-mn`.

Step 2:

We know that `mn` is the sum of all the negative numbers, so now our goal is to find a subset in `ans` whose sum is `-mn`. And then we just need to make all the numbers in the subset negative. We can solve this using a backtracking DFS.

Complexity Analysis

`O(2^N)` time for getting the minimal numbers in `A` and offsetting every number in `A` by `-mn`.

Initiallizing `multiset<int> s` takes `O(2^N * log(2^N)) = O(2^N * N)` time, and `O(2^N)` space.

Within the `for` loop, we at most go through `2^N` elements in `s` and take `O(log(2^N)) = O(N)` time to move/remove an element from the multiset. So each round takes `O(2^N * N)` times. Since we need to repeat `N` times, the overall time complexity is `O(2^N * N^2)`.

The step 2 takes `O(2^N)` time and `O(N)` space, because we have `N` numbers and each of which has two options, inverting or not.

So, overall this algorithm has time complexity `O(2^N * N^2)` and space complexity `O(2^N)`.

Let us first take an example to under the idea:
Suppose n = 4.
So we have elements - 1,2,3,4
There are total n!= 4! = 24 permutations possible. We can see a specific pattern here:

```
arr
[ 1           2           3           4]
1 2 3 4     2 1 3 4     3 1 2 4     4 1 2 3
1 2 4 3     2 1 4 3     3 1 4 2     4 1 3 2
1 3 2 4     2 3 1 4     3 2 1 4     4 2 1 3
1 3 4 2     2 3 4 1     3 2 4 1     4 2 3 1
1 4 2 3     2 4 1 3     3 4 1 2     4 3 1 2
1 4 3 2     2 4 3 1     3 4 2 1     4 3 2 1
So we have 4 block with 6 elements each.

n = 4  we can take an array [1,2,3,4] , initital ans = ""
lets say we have k =15, the 15 th permutation is "3 2 1 4":

As we can see the first value is 3 that means out of the four blocks we need the 3rd block.
Each blocks has n-1! = 3! = 6 elements --> 15 = 6*2 + 3 i.e. we skip 2 blocks and our ans is the third element in the 3rd block
Let us assume the blocks are zero indexed.
Now 15 / 6 = 2;  So we select the 2nd block (0-indexed) that means 2nd index in our array - 3
Now ans = "3"
Remove this element from the array and our array becomes: [1,2,4]
-------------------------------------------------------------------------------------------
```

```
Now we are in this block:
3 1 2 4  - 1
3 1 4 2  - 2   Block 0
  ------
3 2 1 4  - 3 (ans)
3 2 4 1  - 4    Block 1
  ------
3 4 1 2  - 5
3 4 2 1  - 6    Block 2
Now we have 3 blocks each of with 2 elements
i,e. n = n-1 = 3blocks and n-1! = 2! = 2 elements
n = 3, what will be the k? As we passed 12 elements we have k = 15-12 => the third element in this large block.
k = 3
element in partition (p) = 2;
k / p = 3 / 2 = 1 => ans is in block 1, value to add to ans  = 2
arr[1] = 2;
ans = "32"
remove 2 from array => [1,4]
Now we have 2 elements left(n-1 = 3-1)
-----------------------------------------------------------------------------
32  1 4  Block 0
32  4 1  Block 1

n=2, k = 1
1 will be added ans = "321"  arr= [4]
As we only have one value value in array append it to ans.  ans = "3214"

One very important note:(Corner case)
When we have k as a multiple of elements in partition for e.g. k = 12 Then we want to be in block with index 1
but as index = 12 / 6 = 2; we have to keep index = index-1;
Only when we are aiming at the last element we will hit this case.
Here the blocks are zero indexed but the elements inside them are 1 index.

I'm sure after you look at the code you will completely understand it
```

51. N-Queens

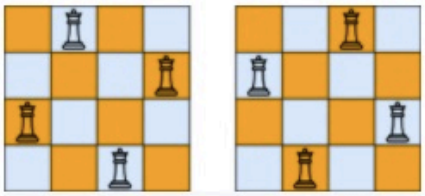
Hard 4801 138 Add to List

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



```
Input: n = 4
Output:
[[".Q..","...Q","Q...","..Q."],
["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two
distinct solutions to the 4-
```

```cpp
class Solution {
public:
    vector<vector<string>> result;
    bool issafe(vector<string> &board, int row, int col){
        for(int i=row;i>=0;i--)
            if(board[i][col]=='Q')
                return false;
        for(int i=row,j=col;i>=0&&j>=0; i--,j--)  //while loop
            if(board[i][j]=='Q') return false;
        for(int i=row,j=col;i>=0&&j<board.size(); i--,j++) //while loop
            if(board[i][j]=='Q') return false;
        return true;
    }

    void dfs(vector<string> &board,int row){
        if(board.size()==row){
            result.push_back(board);
            return;
        }
        for(int i=0;i<board.size();i++){
            if(issafe(board,row,i)){
                board[row][i]='Q';
                dfs(board,row+1);
                board[row][i]='.';
            }
        }


    }
    vector<vector<string>> solveNQueens(int n) {
        vector<string> board(n,string(n,'.'));
        dfs(board,0);
        return result;

    }

};
```

Rat in a Maze Problem - I

Medium Accuracy: 37.73% Submissions: 95375 Points: 4

Consider a rat placed at **(0, 0)** in a square matrix of order **N * N**. It has to reach the destination at **(N - 1, N - 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **'U'(up)**, **'D'(down)**, **'L' (left)**, **'R' (right)**. Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it.
Note: In a path, no cell can be visited more than one time.

Example 1:

```
Input:
N = 4
m[][] = {{1, 0, 0, 0},
         {1, 1, 0, 1},
         {1, 1, 0, 0},
         {0, 1, 1, 1}}
Output:
DDRDRR DRDDRR
Explanation:
The rat can reach the destination at
(3, 3) from (0, 0) by two paths - DRDDRR
and DDRDRR, when printed in sorted order
we get DDRDRR DRDDRR.
```

```
#define vvi vector<vector<int>>
#define vi vector<int>
#define vs vector<string>
class Solution{
    private:
    bool limit(int x,int n){ return (x>=0 and x<n);}
    bool isSafe(int x,int y,int n, vvi visited,vvi & m)
    {
        if( limit(x,n) and limit(y,n) and visited[x][y]==0 and m[x][y]==1) return true;
        else return false;
    }
    void solve(vvi &m,int n,vs &ans,int x,int y,vvi visited,string path)
    {
        if(x==n-1 and y==n-1)
        {
            ans.push_back(path);
            return;
        }
        visited[x][y] = 1;
        // Down
        if(isSafe(x+1,y,n,visited,m))
        {
            path.push_back('D');
            solve(m,n,ans,x+1,y,visited,path);
            path.pop_back();
        }

        //left
        if(isSafe(x,y-1,n,visited,m))
        {
            path.push_back('L');
            solve(m,n,ans,x,y-1,visited,path);
            path.pop_back();
        }

        // right
        if(isSafe(x,y+1,n,visited,m))
        {
            path.push_back('R');
             solve(m,n,ans,x,y+1,visited,path);
            path.pop_back();
        }
        // up
        if(isSafe(x-1,y,n,visited,m))
        {
            path.push_back('U');
             solve(m,n,ans,x-1,y,visited,path);
            path.pop_back();
        }
        //backtrack
        visited[x][y] = 0;
    }

    public:
    vs findPath(vvi &m, int n) {
        vs ans;
        if(!m[0][0])
            return ans;
        vvi visited = m;

        // Initialise visited array with 0
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                visited[i][j] = 0;
        string path = "";
        solve(m,n,ans,0,0,visited,path);
        sort(ans.begin(),ans.end());
        return ans;
    }
};
```

Expected Time Complexity: $O((3^{N^2}))$.
Expected Auxiliary Space: $O(L * X)$, L = length of the path, X = number of paths.

Constraints:
$2 \leq N \leq 5$
$0 \leq m[i][j] \leq 1$

37. Sudoku Solver

Hard 4316 137 Add to List

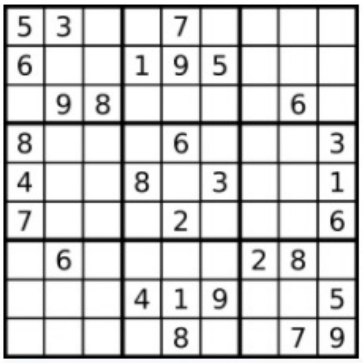
Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

Example 1:



Input: board =

```
class Solution
{
    bool check(vector<vector<char>> &board, int i, int j, char val)
    {
        int row = i - i%3, column = j - j%3;
        for(int x=0; x<9; x++) if(board[x][j] == val) return false;
        for(int y=0; y<9; y++) if(board[i][y] == val) return false;
        for(int x=0; x<3; x++)
            for(int y=0; y<3; y++)
                if(board[row+x][column+y] == val) return false;
        return true;
    }
    bool solveSudoku(vector<vector<char>> &board, int i, int j)
    {
        if(i==9) return true;
        if(j==9) return solveSudoku(board, i+1, 0);
        if(board[i][j] != '.') return solveSudoku(board, i, j+1);

        for(char c='1'; c<='9'; c++)
            if(check(board, i, j, c))
            {
                board[i][j] = c;
                if(solveSudoku(board, i, j+1)) return true;
                board[i][j] = '.';
            }

        return false;
    }
    public:
    void solveSudoku(vector<vector<char>>& board)
    {
        solveSudoku(board, 0, 0);
    }
};
```

886. Possible Bipartition

Medium 1974 49 Add to List

We want to split a group of `n` people (labeled from `1` to `n`) into two groups of **any size**. Each person may dislike some other people, and they should not go into the same group.

Given the integer `n` and the array `dislikes` where `dislikes[i] = [a_i, b_i]` indicates that the person labeled `a_i` does not like the person labeled `b_i`, return `true` *if it is possible to split everyone into two groups in this way*.

Example 1:

Input: n = 4, dislikes = [[1,2],[1,3],[2,4]]
Output: true
Explanation: group1 [1,4] and group2 [2,3].

Example 2:

Input: n = 3, dislikes = [[1,2],[1,3],[2,3]]
Output: false

```
class Solution {
public:
    bool possibleBipartition(int N, vector<vector<int>> &edges)
    {
        vector<vector<int>> adj(N + 1); // adjacency matrix
        vector<int> color(N + 1, 0);
        vector<bool> explored(N + 1, false);

        for (auto &edge: edges)
        {
            adj[edge[0]].push_back(edge[1]);
            adj[edge[1]].push_back(edge[0]);
        }

        queue<int> q;
        for (int i = 1; i <= N; ++i)
            if (!explored[i])
            {
                color[i] = 1;
                q.push(i);
                while (!q.empty())
                {
                    int u = q.front();
                    q.pop();
                    if (explored[u]) continue;

                    explored[u] = true;
                    for (auto v: adj[u])
                    {
                        if (color[v] == color[u]) return false;
                        color[v] = -color[u];
                        q.push(v);
                    }
                }
            }

        return true;
    }
};
```

312. Burst Balloons

Hard 5421 149 Add to List

You are given `n` balloons, indexed from `0` to `n - 1`. Each balloon is painted with a number on it represented by an array `nums`. You are asked to burst all the balloons.

If you burst the `i^th` balloon, you will get `nums[i - 1] * nums[i] * nums[i + 1]` coins. If `i - 1` or `i + 1` goes out of bounds of the array, then treat it as if there is a balloon with a `1` painted on it.

Return *the maximum coins you can collect by bursting the balloons wisely*.

Example 1:

```
Input: nums = [3,1,5,8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] -->
-> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8
+  1*3*8  + 1*8*1 = 167
```

Example 2:

```
Input: nums = [1,5]
Output: 10
```

- The overall idea is **divide and conquer**
  - but we cannot select forward sequentially, because after bursting a balloon, its left and right sides are connected.
  - We first select the last bursted balloon, and then find the entire sequence from back to front one by one.
  - **Once we tried a certain balloon as the last one to be bursted, before it is bursted, its left and right sides are not connected until the left side left a single one and the right side left a single one too.**
    - This is the most important observation to solve this problem.
    - Before you really understand it, you may not find the correct solution easily.
  - We can also modify the code to non-recursive style.

Bottom up DP

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        nums.push_back(1);
        nums.insert(nums.begin(), 1);
        vector<vector<int> > dp(nums.size(), vector<int>(nums.size(), 0));
        for (int i = nums.size() - 3; i >= 0; i --) {
            for (int j = i + 2; j < nums.size();j ++) {
                for (int k = i + 1; k < j; k ++)
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[i] * nums[k] * nums[j]);
            }
        }
        return dp[0][nums.size() - 1];
    }
};
```

Top Down DP

```
int fun(vector<int>&v , int l , int r){
        if(dp[l][r]!=-1){
            return dp[l][r];
        }
        int ans = 0 ;
        for(int i=l ; i<=r ; i++){
            ans = max( ans , v[i]*(l!=0?v[l-1]:1)*(r!=v.size()-1?v[r+1]:1) + (i>0?fun(v,l,i-1):0) + (i<v.size()-1?fun(v,i+1,r):0) );
    }

    return dp[l][r]=ans ;
}
int maxCoins(vector<int>& nums) {
    memset(dp , -1 , sizeof(dp));
    int ans = fun(nums , 0 , nums.size()-1);

    return ans ;
}
```

Reframing the question :-

We are given n balloons indexed from 0 to n-1.Each balloon carry's a number with it representedd by `nums` array . If we burst a balloon we need to multiply the number of that balloon with the neighbouring left and right ballons.

```
For example :-
nums => [2,3,4,5]
say we bursted the 2nd ballon i.e the balloon with number 3
then we need to multiply the left and right neighbours i.e  2 * 3 * 4 which gives us 24 coins .
```

In the end we need to burst all the balloons wisely so that we can get maximum number of coins from them .

- **Edge case:-** The balloons which after bursting creates an out of bound condition then we need to treat out of bound index as a virtual balloon with number 1 .
  For example:-
  In the above example if we burst the balloon with number 2 the left neighbour will go out of bound thus during calculations we will treat the out of bound index as a virtual balloon with number 1 i.e 1[Vritual balloon with out of bound index] * 2 * 3 .

So this completes the entire question .

1st Intuition:-

- After understanding the entire question the first intuition that we may got is backtracking .But after seeing the constraints my eye balls popped out to floor. The reason behind this is that we have `n balloons` so the number of permutaions that will be in the worst case scenerio is O(n!) .
- That's super high . Maybe it will work for constraints till 15 but the question maker has gifted us with a constraint like this `1 <= n <= 500` So in the end say goodbye to backtracking .

2nd Intuition:-

- The sacrifice of backtrack should not go to vein. So after this we all thought of dp but not sure how to implement dp. After some huge brainstorming and taking hints from others I got an observation i.e That the ballons that were already bursted can be ignored for finding the maxcoins for the ballons that we will be bursting in future
- This observation gives us a green signal for using DP's bottom up approach .

3rd Intuition:-

- Now to think for time complexity, we have C(n,k) cases for k balloons and for each case it need to scan the k balloons to compare. Still it's too high but It's better than O(n!) but worse than O(2^n) or maybe O(3^n).

4th Intuition:-

- One thing we all may have understood till now is that there seems to be many sub-similar problems from previous approaches. Somehow if we can cut down those time execution we may get an optimized approach .

5th Intuition:-
i .....k-1....k...k+1..... j
[- - - - - - - - - - - - - - - - - -]

- Till now we are bursting a balloon let's say k'th balloon and we are getting the coins by multiplying the (k-1)th * k'th * (k+1)th balloons . As the balloon k is already burst, we solve the subproblems from i to k -1 and k + 1 to j. But wait, what's going wrong here? The subproblem solve(nums, i, k - 1) and solve(nums, k + 1, j) are not independent since after bursting kth balloon, balloon k - 1 and k + 1 have become adjacent and they will need each other in order to calculate the max profit of coins .
- So in the end we are facing a new isusue in this problem. The issue is that as the left and right become adjacent they will effect the maxCoins in the future calculation . We don't want this thing to happen .

6th Intuition:-

- The above issue will happen in every kth balloon but but there is one exception to this i.e if We choose the kth balloon as the last one to be burst then the subproblems will become independent since (k - 1)th balloon and (k + 1)th balloon won't need each other in order to calculate the answer. WHY? THINK MY FRIENDS!!!

7th Intuition:-

- So to answer the 6th Intuition, if we think with a pen and paper we get that becuase only the first and last balloons are the balloons where we are sure of their adjacent balloons before hand! Because
  For the first we have nums[k-1] * nums[k] * nums[k+1] and for the last we have nums[-1] * nums[k] * nums[n] .
- OKkk. Now think about n balloons if j is the last one to burst, We can see that the balloons are again getting separated into 2 sections. But this time since the balloon j is the last balloon of all to burst, the left and right section now has well defined boundary and do not affect each other!
- Thus now we can do either recursive with memoization or our favourite dp .

Final Intuition
Let's recap a bit:

- **Edge case:** Pad the beginning + end of the array with a virtual 1, since the problem defines it this way, it won't affect the final value, and most importantly it eliminates the need to deal with these special cases
- Realized that **working backwards** will allow us to **cleanly divide the array** into subproblems
- **Pop all the 0 balloons first** and remove them from the array (since they are worth nothing)
- Now **there are 3 variables** in our main equation: the values of the 3 balloons to pop. Again, we use the two 1's we just padded the array with to eliminate two of those variables off the bat
- Now just **try all the possible middle balloons to pop** (the 3rd variable). For each balloon we choose, use it as the right and left ballon of the next level of recursion, along with the padded 1's, and so forth.
- **Base case** is when there are no more balloons between the left and right balloon indexes (left+1 == right)
- **Time complexity** will be `O(N^3)`

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        //including the nums[-1] and nums[n]
        int n = nums.size() + 2;
        vector<vector<int>> dp(n, vector<int>(n));
        vector<int> new_nums(n, 1);
        int i = 1;
        for(auto num : nums) {
            new_nums[i++] = num;
        }
        for(int len = 2; len <= n; len++) {
            //iterate from interval length from 2 to n
            for(int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                //select between left and right boundary (i, j)
                for(int k = i + 1; k < j; k++) {
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + new_nums[i] * new_nums[k] * new_nums[j]);
                }
            }
        }
        return dp[0][n - 1];
    }
};
```

131. Palindrome Partitioning

Medium 5758 173 Add to List

Given a string `s`, partition `s` such that every substring of the partition is a **palindrome**. Return all possible palindrome partitioning of `s`.

A **palindrome** string is a string that reads the same backward as forward.

Example 1:

```
Input: s = "aab"
Output: [["a","a","b"],
["aa","b"]]
```

Example 2:

```
Input: s = "a"
Output: [["a"]]
```

Reframing the question:-
Find all the substrings which satisfies the palindrome condition of given string s.

Intuition:-

- The structure of the question is quite clear that it **promotes the use of recursion** . The recursion approach of this question can be like: we start iteration from beginning and step by step we increase the size of substr that we are checking for palindromes . Once the palindrome substr is found ,the substr needs to go to a vector/list storing those palindromes substr . This is how the brute recursion will work .
- The **optimization** of the above recursion is **backtracking**.
  A **backtracking algorithm uses the depth-first search method**. When it starts exploring the solutions, a bounding or a helper function is applied so that the **algorithm can check if the so-far built solution satisfies the constraints**. If it does, it continues searching. If it doesn't, the branch would be eliminated, and the algorithm goes back to the level before. In simple words backtracking starts from some intermediate position where there is still a hope to get a new palindrome .
- In backtracking algo there is **always a helper metod** or a safe checker merhod which checks whether the built soution satisifes the conditions or not . This one thing has **certainity of 100%**.

Algorithm :-

1. As i told you in **backtracking,** one thing is sure i.e a **helper method**. In our problem we will need a bool helper method which will check the palindrome conditions, let's name it as `bool isPalindrome()` .
2. Now let's **create some variables** : `result` vector for storing all the substring which satisfies the palindrome condition , `path` vector for storing the ongoing substr which has the potential of satisfying the conditions of palindrome , `start` variable for storing the start position of the current substr , `index i` for traversing inside the substring and `n` fro storing the size of given string `s` .
3. **Now we search** from `start` till the end of the string. Once we reach a position `i` such that the sub-string from `start to i (s.substr(start, i - start + 1))` is `a palindrome` , we add it to our `path` variable. Then we recursively call the same method to execute the remaining substring . Once we reach the end of the string, we add palindromes `path` into the `result` of all the *possible partitioning* .
4. Remember that at position `i` , we find `s.substr(start, i - start + 1)` to be a palindrome and we immediately add it to `path` . Now think that there may be a position `j` such that `j > i` and `s.substr(start, j - start + 1)` is also a palindrome. Thus now we need to go back to our `start` before adding `s.substr(start, i - start + 1)` to `path` and continue to find the next palindrome position after `i` . And after this evertime we simply need to remove or pop `s.substr(start, i - start + 1)` out of `path` to execute the backtracking algo .
5. The steps we executed in the above step is none other than the **famous dfs** .

Code:-

```cpp
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;
        partition(s, 0, path, result);//dfs calls
        return result;
    }
private:
    //DFS steps
    void partition(string& s, int start, vector<string>& path, vector<vector<string>>& result) {
        int n = s.length();
        if (start == n) {
            result.push_back(path);
        } else {
            for (int i = start; i < n; i++) {
                if (isPalindrome(s, start, i)) {
                    path.push_back(s.substr(start, i - start + 1));
                    partition(s, i + 1, path, result);
                    path.pop_back();
                }
            }
        }
    }
    //helper function to safe check whether a substr is palindrome or not
    bool isPalindrome(string& s, int l, int r) {
        while (l < r) {
            if (s[l++] != s[r--]) {
                return false;
            }
        }
        return true;
    }
};
```

Time Complexity :- `O(n*(2^n)) [n=length of the string]`
Space Complexity :- `O(n) [beacause of recursion stack]`

Minimum Score triangulation of a polygon Similar to Matrix Multiplication

```
class Solution {
public:
    int dp[1002][1002];
    int solve(vector<int> arr, int i,int  j){
        if(i>=j) return 0;

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

        int ans = INT_MAX;
        for(int k=i;k<j;k++){

            int temp = solve(arr,i,k) + solve(arr,k+1,j) + (arr[i-1] * arr[k] * arr[j]);
            ans = min(ans,temp);
        }
        return dp[i][j] = ans;
    }
    int minScoreTriangulation(vector<int>& values) {
        memset(dp,-1,sizeof(dp));
        return solve(values,1,values.size()-1);
    }
};
```

```
int mergeStones(vector<int>& stones, int K) {
    int n = stones.size();
    if ((n - 1) % (K - 1)) return -1;

    vector<int> prefix(n + 1);
    for (int i = 0; i <  n; i++)
        prefix[i + 1] = prefix[i] + stones[i];

    vector<vector<int> > dp(n, vector<int>(n));
    for (int m = K; m <= n; ++m)
        for (int i = 0; i + m <= n; ++i) {
            int j = i + m - 1;
            dp[i][j] = INT_MAX;
            for (int mid = i; mid < j; mid += K - 1)
                dp[i][j] = min(dp[i][j], dp[i][mid] + dp[mid + 1][j]);
            if ((j - i) % (K - 1) == 0)
                dp[i][j] += prefix[j + 1] - prefix[i];
        }
    return dp[0][n - 1];
}
```

FAQ

Q: Why `mid` jump at step `K - 1`
A: We can merge `K` piles into one pile,
we can't merge `K + 1` piles into one pile.
We can merge `K + K - 1` piles into on pile,
We can merge `K + (K - 1) * steps` piles into one pile.

Update 2019-03-04
Sorry guys, It seems that somehow it started kind of debate in one of my replies.
I didn't mean to do that and I feel I have to say something.

1. Yes, I agree that people have right to express in their comfortable language, including Chinese.
   It's not the same as the situation of a meeting room.
   User don't take others' time and force them to listen to you.
   Reader can choose what they want to read.
   Like ebooker and trip adviser, they have comments in all languages.

2. I strongly disagree any unreasonable downvotes.
   Posts and reply should not be downvoted for no reason like language.
   Personally I receive downvotes for each of my posts.
   Of course, people have right to do that but please at least say something or leave a message.
   Like "I downvote for the reason that....", so that I can also improve somehow, right?

3. I encourage everyone to express in English and discuss with all others.
   The main reason is that English is still one important skill for engineers.
   We need to learn from documents in English.
   Moreover, as a Chinese engineer, I hope I can bring the good stuff back to the world.

4. In the end, the most important, I encourage everyone to learn some Chinese :)

22. Generate Parentheses

Medium 11132 436 Add to List Share

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

```cpp
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> res;
        addingpar(res, "", n, 0);
        return res;
    }
    void addingpar(vector<string> &v, string str, int n, int m){
        if(n==0 && m==0) {
            v.push_back(str);
            return;
        }
        if(m > 0){ addingpar(v, str+")", n, m-1); }
        if(n > 0){ addingpar(v, str+"(", n-1, m+1); }
    }
};
```

Coin Change | DP-7

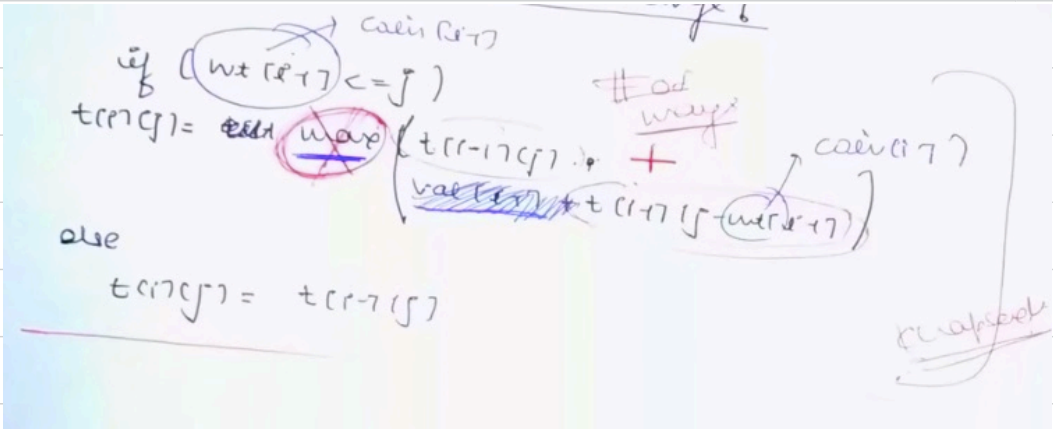
Difficulty Level : Hard • Last Updated : 17 Nov, 2021

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.
For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2}, {1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

Variation of Unbounded Knapsack

Either we will not select or we will select previous

Add both to get the current number of choices




```
long getNumberOfWays(long N, vector<long> Coins)
{

    // Create the ways array to 1 plus the amount
    // to stop overflow
    vector<long> ways(N + 1);

    // Set the first way to 1 because its 0 and
    // there is 1 way to make 0 with 0 coins
    ways[0] = 1;

     // Go through all of the coins
    for(int i = 0; i < Coins.size(); i++)
    {

        // Make a comparison to each index value
        // of ways with the coin value.
        for(int j = 0; j < ways.size(); j++)
        {
            if (Coins[i] <= j)
            {

                // Update the ways array
                ways[j] += ways[(j - Coins[i])];
            }
        }
    }

    // Return the value at the Nth position
    // of the ways array.
    return ways[N];
}
```

We have taken a vector of coins numbered 1,2,..., N

Declare a DP vector of size(N+1) and name it ways and set ways[0] as 1 as number of ways to select 1 rs coin with 1 rs coin is 1
Now start your for loop and it will go will your coin vector size i.e. N.
Inside the for loop you will have one other for loop which will go from 0 to ways array size.

Now just like knapsack, check

If the coins[I] <=j
Then in that case update the way array by ways[j-coins[I]] just like the way you used to do in the knapsack

Here it is unbounded knapsack so you will not have any value array. Only weight array.

Rest is same. Return ways[n]

139. Word Break

Medium 9173 412 Add to List Share

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

The DP aprroach used is top to bottom. The idea is, if DP[i] = true, meaning that it is possible to create substring [0,i], then what all substrings can we create from here using the words in dictionary. So, DP[i + 1 + word.size()-1] = true , if word is in dictinonary and it is a substring in the main string starting from index i+1.
At the end if DP[n-1] = true, then we can create the complete string using some strings from the dictionary.
The DP had to be initialised with string in dictionary which are substrings in the main string starting from index 0.
Note: there can be optimisations made for prefix search starting from i+1 instead of find().
The solution:

```cpp
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        int n = s.size();
        vector<bool> dp(n+1, false);
        for(int j = 0; j<wordDict.size(); j++){
            size_t pos = s.find(wordDict[j], 0);
            if(pos != string::npos && pos == 0){
                dp[wordDict[j].size()-1] = true;
            }
        }

        for(int i=0; i<n; i++){
            if(dp[i]){
                for(int j = 0; j<wordDict.size(); j++){
                    size_t pos = s.find(wordDict[j], i+1);
                    if(pos != string::npos && pos == i+1){
                        if(i+wordDict[j].size() == n-1) return true;
                        dp[i+wordDict[j].size()] = true;
                    }
                }
            }
        }
        return dp[n-1];
    }
};
```

Maximum path sum in a matrix

```cpp
class Solution{
public:
    int maximumPath(int N, vector<vector<int>> Matrix)
    {

       int dp[N][N];
       for(int i=0;i<N;i++)
           for(int j=0;j<N;j++){
               if(i==0) dp[i][j]=Matrix[i][j];
               else{
                   if(j==0)
                       dp[i][j]=Matrix[i][j]+max(dp[i-1][j],dp[i-1][j+1]);
                   else if(j==N-1)
                   dp[i][j]=Matrix[i][j]+max(dp[i-1][j-1],dp[i-1][j]);
                   else
                       dp[i][j]=Matrix[i][j]+
                       max({dp[i-1][j-1],dp[i-1][j],dp[i-1][j+1]});
               }
           }
       int ans=dp[N-1][0];
       for(int j=1;j<N;j++)
           if(dp[N-1][j] > ans)
               ans=dp[N-1][j];
       return ans;
   }
```

Longest path in a matrix

```cpp
class Solution {
public:
    int traverse(vector<vector<int>> &mat,int i,int
            j,vector<vector<int>> &dp,int &ans)
    {
        if(dp[i][j]!=0) return dp[i][j];
        int n=mat.size(),m=mat[0].size();
        int a1[]={-1,1,0,0};
        int a2[]={0,0,-1,1};
        for(int k=0;k<4;k++)
        {
            int a=i+a1[k],b=j+a2[k];
            if(a>=0&&a<n&&b>=0&&b<m)
            {
                if(mat[a][b]>mat[i][j])
                    dp[i][j]=max(dp[i][j],
                1+traverse(mat,a,b,dp,ans));
            }
        }
        ans=max(ans,dp[i][j]);
        return dp[i][j];
    }
    int longestIncreasingPath(vector<vector<int>>& mat)
    {
        int ans=0,n=mat.size(),m=mat[0].size();
        vector<vector<int>> dp(n,vector<int>(m,0));
        for(int i=0;i<n;i++)
            for(int j=0;j<m;j++)
                if(dp[i][j]==0)
                    traverse(mat,i,j,dp,ans);
        return ans+1;
    }
};
```

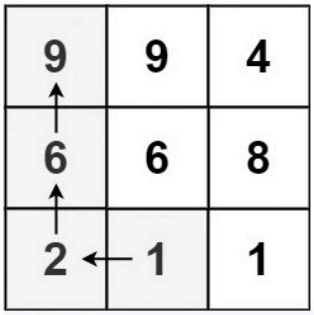
329. Longest Increasing Path in a Matrix

Hard 4661 82 Add to List Share

Given an `m x n` integers `matrix`, return *the length of the longest increasing path in* `matrix`.

From each cell, you can either move in four directions: left, right, up, or down. You **may not** move **diagonally** or move **outside the boundary** (i.e., wrap-around is not allowed).

Example 1:




```
Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4
Explanation: The longest increasing path
is [1, 2, 6, 9].
```

```cpp
#define vvi vector<vector<int>>
#define MIN INT_MIN
#define MAX INT_MAX
class Solution {
public:
    int f(int r, int c,vvi& M, vvi& dp )
    {
        int n = M.size(), m = M[0].size();
        if(r>=n or r<0 or c>=m or c<0) return 0;
        if(dp[r][c]!=-1) return dp[r][c];
        int left=MIN,right=MIN,up=MIN,down=MIN;
        if(c>0 and M[r][c]<M[r][c-1]) left = 1+f(r,c-1,M,dp);
        if(c+1<M[0].size() and M[r][c] < M[r][c+1]) right = 1+ f(r,c+1,M,dp);
        if(r>0 and M[r][c] < M[r-1][c]) up = 1+f(r-1,c,M,dp);
        if(r+1<M.size() and M[r][c] < M[r+1][c]) down = 1+f(r+1,c,M,dp);
        return dp[r][c] = max({1,right,left,up,down});
    }
    int longestIncreasingPath(vector<vector<int>>& M)
    {
        int n = M.size();
        int m = M[0].size();
        int result=0;
        vector<vector<int>> dp(n,vector<int>(m,-1));
        for(int i=0;i<n;i++) {
            for(int j=0;j<m;j++)
            {
                f(i,j,M,dp);
                result = max(result,dp[i][j]);
            }
        }
        return result;
    }
};
```

The solution expects your hold on recursion and recursive calls
Dont worry if you dont know how recursion works, you must have faith on recursion fairy which does all the work like magic, you just need to think the base cases and let the fairy handle all the intricacies.

Let us understand how the code works

- Look at the longestIncreasingPath function, it has input parameters as 2d vector M
- number of rows = n and number of cols = m which we computed as M.size() and M[0].size() resp.
- We will store the result of the longest increasing path in the integer variable result. Declare it.
- Now declare a 2D DP vector having n rows and m cols initialised with -1 in all cells.
- Now run a for loop for all rows and for all cols and now run the helper function f which will make use of multiple recursion calls.
- After calling the helper function f, we will update the maximum of result and dp[i][j] in the 2d vector DP.
- At the end we will return the result

Let us now understand this recursive function f

- So the helper function f takes number of rows, number of columns, MATRIX(call by ref) and DP(call by ref)
- The base cases of this helper function will be if your row exceeds or equal to n or row is less than 0 or columns exceed or equal to m or columns less than 0 then the helper function will return 0.
- We will have four variables left, right, up and down and preinitialised with INT_MIN
- now to avoid unnecessary recursive calls, if a cell of dp[i][j] is not equal to -1 that means, its value has been updated and already present in the memo table DP, then we will return that, that is the significance of the IF block.
- Now if number of columns are greater than 0 and value in current row current column is less than value of matrix current row previous column, then in this case we will update LEFT to be 1+ f(r,c-1,M,dp) as left cell is having more value, so we will update the left value and add 1 to it.
- Likewise we will check if right cell i.e. M[r][c+1] is greater than M[r][c] then update right as 1+f(r,c+1,M,dp)
- Do the same thing for up and down and return maximum of 1,left, right, up and down.

Word Break

Medium Accuracy: 50.24% Submissions: 25991 Points: 4

Given a string **A** and a dictionary of n words **B**, find out if A can be segmented into a space-separated sequence of dictionary words.

Note: From the dictionary **B** each word can be taken any number of times and in any order.

Example 1:

```
Input:
n = 12
B = { "i", "like", "sam",
"sung", "samsung", "mobile",
"ice","cream", "icecream",
"man", "go", "mango" }
A = "ilike"
Output:
1
Explanation:
The string can be segmented as "i like".
```

```cpp
class Solution
{
public:
    int wordBreak(string A, vector<string> &dict)
    {
        int n = A.length();
        int dp[n+1];
        memset(dp, 0, sizeof(dp));
        dp[0] = 1;
        for(int i=1; i<=n; i++){
            for(int j=0; j<i; j++)
            {
                //inner loop to form each word from string A
                if(dp[j])
                {
                    //if dp[j] i.e. if the word is possible
                    string sub = A.substr(j, i-j);
                    // calculate the remaining string from j to i-j
                    auto it = find(dict.begin(), dict.end(), sub);
                    //find the reamining word present in dict
                    if(it!=dict.end())
                    {
                        //if present
                        dp[i] = 1;      //String is possible
                        break;
                    }
                }
            }
        }
        return dp[n];
    }
};
```

vi dp(n+1, 0) ;

Word Break - Part 2

Hard Accuracy: 63.66% Submissions: 6141 Points: 8

Given a string **s** and a dictionary of words **dict** of length **n,** add spaces in **s** to construct a sentence where each word is a valid dictionary word. Each dictionary word can be used more than once. Return all such possible sentences.

Follow examples for better understanding.

Example 1:

```
Input: s = "catsanddog", n = 5
dict = {"cats", "cat", "and", "sand", "dog"}
Output: (cats and dog)(cat sand dog)
Explanation: All the words in the given
sentences are present in the dictionary.
```

Example 2:

```
Input: s = "catsandog", n = 5
dict = {"cats", "cat", "and", "sand", "dog"}
Output: Empty
Explanation: There is no possible breaking
of the string s where all the words are present
in dict.
```

Your Task:

You do not need to read input or print anything. Your task is to complete the function **wordBreak()** which takes **n**, **dict** and **s** as input parameters and returns a list of possible sentences. If no sentence is possible it returns an empty list.

Expected Time Complexity: $O(N^2*n)$ where N = |s|

Expected Auxiliary Space: $O(N^2)$

```cpp
class Solution
{
    public:vector<string> ans;
    void solve(unordered_map<string,int> mp,string s,string str)
    {
        if(s.length()==0)
        {
            ans.push_back(str);
            return;
        }
        for(int i=0;i<s.length();i++)
        {
            string left=s.substr(0,i+1);
            if(mp.find(left)!=mp.end())
            {
                string right=s.substr(i+1);
                if(i<s.length()-1) left+=" ";
                solve(mp,right,str+left);
            }
        }

    }
    vector<string> wordBreak(int n, vector<string>& dict, string s)
    {
        string str="";
        ans.clear();
        unordered_map<string,int> mp;
        for(int i=0;i<n;i++)
            mp[dict[i]]++;
        solve(mp,s,str);
        return ans;
    }
};
```

anddogs. Cats_

base case

Catsanddog

left

true

anddog

left="cats."

insert into map

pass map into fn

Substring - Subsequence problem

Medium Accuracy: 42.54% Submissions: 564 Points: 4

Find the longest subsequence X of a string A which is a substring Y of a string B.

Note: All letters of the Strings are Uppercased.

Example 1:

```
Input:
A = "ABCD" , B = "BACDBDCD"
Output:
3
Explanation:
The answer would be 3 as because "ACD"
is the longest subsequence of A which
is also a substring of B.
```

Example 2:

```
Input:
A = "A" , B = "A"
Output:
1
Explanation:
The answer would be 1 as because "A"
is the longest subsequence of A which
is also a substring of B.
```

```cpp
class Solution {
  public:
    int getLongestSubsequence(string x, string y) {
        // code here
        int m=x.size(),n=y.size();
        int dp[m+1][n+1];

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

Edit Distance

Medium Accuracy: 49.98% Submissions: 48500 Points: 4

Given two strings **s** and **t**. Return the minimum number of operations required to convert **s** to **t**.
The possible operations are permitted:

1. Insert a character at any position of the string.
2. Remove any character from the string.
3. Replace any character from the string with any other character.

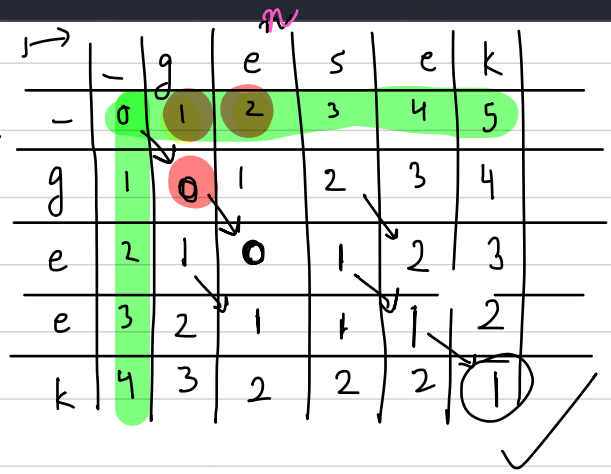
Example 1:

```
Input:
s = "geek", t = "gesek"
Output: 1
Explanation: One operation is required
inserting 's' between two 'e's of str1.
```

Example 2:

```
Input :
s = "gfg", t = "gfg"
Output:
0
Explanation: Both strings are same.
```

```cpp
class Solution {
  public:
    int editDist(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;
                else if (j == 0)
                    dp[i][j] = i;
                else if (str1[i - 1] == str2[j - 1])
                    dp[i][j] = dp[i - 1][j - 1];
                else
                    dp[i][j] = 1 + min( min ( dp[i][j-1],dp[i-1][j]) ,dp[i - 1][j - 1] ) ;
            }
        }
        return dp[m][n];
    }
    int editDistance(string s, string t){
        return editDist(s,t,s.size(),t.size());
    }
};
```



Space optimised edit distance

```
#include <bits/stdc++.h>S
using namespace std;
int minDis(string s1, string s2, int n, int m, vector<vector<int>> &dp){

  // If any string is empty,
  // return the remaining characters of other string

  if(n==0)    return m;

  if(m==0)    return n;


  // To check if the recursive tree
  // for given n & m has already been executed

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


  // If characters are equal, execute
  // recursive function for n-1, m-1

  if(s1[n-1]==s2[m-1]){
    if(dp[n-1][m-1]==-1){
      return dp[n][m] = minDis(s1, s2, n-1, m-1, dp);
    }
    else
      return dp[n][m] = dp[n-1][m-1];
  }


  // If characters are nt equal, we need to

  // find the minimum cost out of all 3 operations.

  else{
    int m1, m2, m3;        // temp variables

    if(dp[n-1][m]!=-1){
      m1 = dp[n-1][m];
    }
    else{
      m1 = minDis(s1, s2, n-1, m, dp);
    }

    if(dp[n][m-1]!=-1){
      m2 = dp[n][m-1];
    }
    else{
      m2 = minDis(s1, s2, n, m-1, dp);
    }

    if(dp[n-1][m-1]!=-1){
      m3 = dp[n-1][m-1];
    }
    else{
      m3 = minDis(s1, s2, n-1, m-1, dp);
    }
    return dp[n][m] = 1+min(m1, min(m2, m3));
  }

}
```

5. Longest Palindromic Substring

Medium 15371 901 Add to List Share

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

```
#define vi vector<int>
#define vvi vector<vi>
class Solution {
public:
    int lcs(string s, string t){
        vvi dp(s.size()+1,vi(t.size()+1,0));
        for (int i=1; i<=s.size();i++) for (int j=1;j<=t.size();j++)
                dp[i][j] = s[i-1]==t[j-1] ? 1+dp[i-1][j-1] : max(dp[i-1][j],dp[i][j-1]);
        return dp[s.size()][t.size()];
    }
    int longestPalindromeSubseq(string s) {
        string s1 = s;
        reverse(s.begin(),s.end());
        return lcs(s,s1);

    }
};
```

2 pointer approach

```
class Solution {
public:
    string longestPalindrome(string str) {
        string ans;
        int max_len = 0;
        int start, end;
        for(int i=0;i<str.length();i++){
            int l = i, r = i;
            while(l>=0 && r<str.length() && str[l]==str[r]){
                if((r-l+1) > max_len){
                    start = l;
                    max_len = r-l+1;
                }
                l--;r++;
            }
            l = i; r = i+1;
            while(l>=0 && r<str.length() && str[l]==str[r]){
                if((r-l+1) > max_len){
                    start = l;
                    max_len = r-l+1;
                }
                l--;r++;
            }
        }
        return str.substr(start, max_len);
    }
};
```

Minimum sum partition

Hard Accuracy: 50.0% Submissions: 36063
Points: 8

Given an integer array **arr** of size **N**, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum and find the minimum difference

Example 1:

```
Input: N = 4, arr[] = {1, 6, 11, 5}
Output: 1
Explanation:
Subset1 = {1, 5, 6}, sum of Subset1
Subset2 = {11}, sum of Subset2 = 11
```

Example 2:

```
Input: N = 2, arr[] = {1, 4}
Output: 3
Explanation:
Subset1 = {1}, sum of Subset1 = 1
Subset2 = {4}, sum of Subset2 = 4
```

```cpp
#define vi vector<int>
#define vvi vector<vi>
class Solution{
  public:
    iint f(int A[],int n,int sum,int s,vvi &dp){
        if(!n) return abs(sum-2*s);
        if(dp[n][s]!=-1) return dp[n][s];
        return dp[n][s]=min(f(A,n-1,sum,s+A[n-1],dp),f(A,n-1,sum,s,dp));
    }
    int minDifference(int A[], int n)  {
        int sum=0;
        for(int i=0;i<n;i++) sum+=A[i];
        vvi dp(n+1,vi(sum+1,-1));
        return f(A,n,sum,0,dp);
    }
};
```

Count number of hops

Easy Accuracy: 46.41% Submissions: 30832 Points: 2

A frog jumps either 1, 2, or 3 steps to go to the top. In how many ways can it reach the top. As the answer will be large find the answer modulo 1000000007.

Example 1:

```
Input:
N = 1
Output: 1
```

Example 2:

```
Input:
N = 4
Output: 7
Explanation:Below are the 7 ways to reach
4
1 step + 1 step + 1 step + 1 step
1 step + 2 step + 1 step
2 step + 1 step + 1 step
1 step + 1 step + 2 step
2 step + 2 step
3 step + 1 step
1 step + 3 step
```

Your Task:
Your task is to complete the function **countWays()** which takes 1 argument(N) and returns the answer%(10^9 + 7).

Expected Time Complexity: O(N).
Expected Auxiliary Space: O(1).

Constraints:
$1 \le N \le 10^5$

```cpp
class Solution
{
   public:

   long long countWays(int n)
    {
        long long mod = 1e9 + 7;

        vector<long long> dp(n+1,0);
        dp[0] = 1;
        dp[1] = 1;
        dp[2] = 2;
        for(int i=3; i<n+1; i++) dp[i] = (dp[i-1] + dp[i-2] + dp[i-3])%mod;
        return dp[n];
    }
};
```

Egg Dropping Puzzle

Medium Accuracy: 54.38% Submissions: 33469 Points: 4

You are given **N** identical eggs and you have access to a **K**-floored building from **1** to **K**.

There exists a floor **f** where **0** <= **f** <= **K** such that any egg dropped at a floor higher than **f** will break, and any egg dropped **at or below** floor **f** will **not break**. There are few rules given below.

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If the egg doesn't break at a certain floor, it will not break at any floor below.
- If the eggs breaks at a certain floor, it will break at any floor above.

Return the minimum number of moves that you need to determine with certainty what the value of **f** is.

For more description on this problem see wiki page

Example 1:

```
Input:
N = 1, K = 2
Output: 2
Explanation:
1. Drop the egg from floor 1. If it
   breaks, we know that f = 0.
2. Otherwise, drop the egg from floor 2.
   If it breaks, we know that f = 1.
3. If it does not break, then we know f = 2.
4. Hence, we need at minimum 2 moves to
   determine with certainty what the value of f is.
```

Expected Time Complexity : O(N*(K^2))
Expected Auxiliary Space: O(N*K)

The following is a description of the instance of this famous puzzle involving N=2 eggs and a building with H=36 floors:[13]

> Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing (using U.S. English terminology, in which the first floor is at ground level). We make a few assumptions:
>
> - An egg that survives a fall can be used again.
> - A broken egg must be discarded.
> - The effect of a fall is the same for all eggs.
> - If an egg breaks when dropped, then it would break if dropped from a higher window.
> - If an egg survives a fall, then it would survive a shorter fall.
> - It is not ruled out that the first-floor windows break eggs, nor is it ruled out that eggs can survive the 36th-floor windows.
>
> If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second-floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the lowest number of egg-droppings that is guaranteed to work in all cases?

To derive a dynamic programming functional equation for this puzzle, let the **state** of the dynamic programming model be a pair s = (n,k), where

n = number of test eggs available, $n = 0, 1, 2, 3, \ldots, N-1$.

k = number of (consecutive) floors yet to be tested, $k = 0, 1, 2, \ldots, H-1$.

For instance, $s = (2,6)$ indicates that two test eggs are available and 6 (consecutive) floors are yet to be tested. The initial state of the process is $s = (N,H)$ where N denotes the number of test eggs available at the commencement of the experiment. The process terminates either when there are no more test eggs ($n = 0$) or when $k = 0$, whichever occurs first. If termination occurs at state $s = (0,k)$ and $k > 0$, then the test failed.

Now, let

$W(n,k)$ = minimum number of trials required to identify the value of the critical floor under the worst-case scenario given that the process is in state $s = (n,k)$.

Then it can be shown that[14]

$$W(n,k) = 1 + \min\{\max(W(n-1, x-1), W(n,k-x)): x = 1, 2, \ldots, k\}$$

with $W(n,0) = 0$ for all $n > 0$ and $W(1,k) = k$ for all k. It is easy to solve this equation iteratively by systematically increasing the values of n and k.

Faster DP solution using a different parametrization [edit]

Notice that the above solution takes $O(nk^2)$ time with a DP solution. This can be improved to $O(nk \log k)$ time by binary searching on the optimal x in the above recurrence, since $W(n-1,x-1)$ is increasing in x while $W(n,k-x)$ is decreasing in x, thus a local minimum of $\max(W(n-1,x-1), W(n,k-x))$ is a global minimum. Also, by storing the optimal x for each cell in the DP table and referring to its value for the previous cell, the optimal x for each cell can be found in constant time, improving it to $O(nk)$ time. However, there is an even faster solution that involves a different parametrization of the problem:

Let k be the total number of floors such that the eggs break when dropped from the kth floor (The example above is equivalent to taking $k = 37$).

Let m be the minimum floor from which the egg must be dropped to be broken.

Let $f(t,n)$ be the maximum number of values of m that are distinguishable using t tries and n eggs.

Then $f(t,0) = f(0,n) = 1$ for all $t, n \geq 0$.

Let a be the floor from which the first egg is dropped in the optimal strategy.

If the first egg broke, m is from 1 to a and distinguishable using at most $t-1$ tries and $n-1$ eggs.

If the first egg did not break, m is from $a+1$ to k and distinguishable using $t-1$ tries and n eggs.

Therefore, $f(t,n) = f(t-1,n-1) + f(t-1,n)$.

Then the problem is equivalent to finding the minimum x such that $f(x,n) \geq k$.

To do so, we could compute $\{f(t,i) : 0 \leq i \leq n\}$ in order of increasing t, which would take $O(nx)$ time.

Thus, if we separately handle the case of $n = 1$, the algorithm would take $O(n\sqrt{k})$ time.

But the recurrence relation can in fact be solved, giving $f(t,n) = \sum_{i=0}^{n} \binom{t}{i}$, which can be computed in $O(n)$ time using the identity $\binom{t}{i+1} = \binom{t}{i} \frac{t-i}{i+1}$ for all $i \geq 0$.

Since $f(t,n) \leq f(t+1,n)$ for all $t \geq 0$, we can binary search on t to find x, giving an $O(n \log k)$ algorithm. [15]

```cpp
class Solution
{
    public:
   //Function to find minimum number of attempts needed in
   //order to find the critical floor.
   int eggDrop(int n, int k)
   {
       // your code here
       if(n==1)
           return k;
       if(k==1)
           return 1;
       int mid = (k+1)/2;
       if((k-mid) > (mid-1))
           return 1+eggDrop(n, k-mid);
       else
           return 1+eggDrop(n-1, mid-1);
   }
};
```

This is a classic Egg dropping problem

```
/*
The basic idea is, we have to find the minimum number of attempts to find the
Threshold Floor, that means, above that floor the egg will break and below that
floor the egg will not break

So, we have two choices
1. Egg will break
2. Egg will not break

-> For the first case if the egg will break from a particular floor then we have to go below
that floor
-> For the second case if the egg will not break from a particular floor then we have to go
above to find the threshold floor

Base Conditions:
1. It is given that egg the will not be 0 so no check for this in code
2. If the egg will be 1 i.e. k = 1, So we need at minimum n moves to determine what the value
    of floor is i.e n attempts.
3. If the n is 0 then no floor 0 attempt.
4. If the n is 1 then only 1 attempt.
*/
```

ii. Recursive + Memoized TopDown, It will also give TLE

```cpp
class Solution {
public:
    //Time: O(n^2*k), Space: O(n*k)
    int helper(int k, int n, vector<vector<int>>& memo){
        if(n == 0 || n == 1) return n;
        if(k == 1) return n;

        if(memo[k][n] != -1) return memo[k][n];

        int mn = INT_MAX;

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

            /*representing both the choices with memo
            First one, if the egg will break, no. of eggs will decreased and we have to
            down from that floor.
            Second one, if the egg will not break, no. of eggs will not decreased and we
            have to go above form that floor.*/
            int temp = 1 + max(helper(k-1, i-1, memo), helper(k, n-i, memo));

            mn = min(mn, temp); //minimum number of attempts
        }
        return memo[k][n] = mn;
    }

    int superEggDrop(int k, int n) {
        //k means number of eggs, n means number of floors
        vector<vector<int>> memo(k+1, vector<int>(n+1, -1));
        return helper(k, n, memo);
    }
};
```

i. Recursive Solution, It will give TLE

```cpp
class Solution {
public:
    //Time: O(n*(2^min(n,k))), Space: O(n)
    int helper(int k, int n){
        if(n == 0 || n == 1) return n;
        if(k == 1) return n;

        int mn = INT_MAX;

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

            /*representing both the choices
            First one, if the egg will break, no. of eggs will decreased and we have to
            down from that floor.
            Second one, if the egg will not break, no. of eggs will not decreased and we
            have to go above form that floor.*/
            int temp = 1 + max(helper(k-1, i-1), helper(k, n-i));

            mn = min(mn, temp);  //minimum number of attempts
        }
        return mn;
    }

    int superEggDrop(int k, int n) {
        //k means number of eggs, n means number of floors
        return helper(k, n);
    }
};
```

iii. Optimized Memoization, It will also give TLE

```
class Solution {
public:
    //Time: O(n^2*k), Space: O(n*k)
    int helper(int k, int n, vector<vector<int>>& memo){
        if(n == 0 || n == 1) return n;
        if(k == 1) return n;

        if(memo[k][n] != -1) return memo[k][n];

        int mn = INT_MAX;

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

            /*here is the optimization goes on, the basic idea is before
            any function call we are checking if any value from that function
            call is present or not*/
            int left = 0, right = 0;
            if(memo[k-1][i-1] != -1)
                left = memo[k-1][i-1];
            else{
                left = helper(k-1, i-1, memo);
                memo[k-1][i-1] = left;
            }

            if(memo[k][n-i] != -1)
                right = memo[k][n-i];
            else{
                right = helper(k, n-i, memo);
                memo[k][n-i] = right;
            }

            int temp = 1 + max(left, right);

            mn = min(mn, temp); //minimum number of attempts
        }
        return memo[k][n] = mn;
    }

    int superEggDrop(int k, int n) {
        //k means number of eggs, n means number of floors
        vector<vector<int>> memo(k+1, vector<int>(n+1, -1));
        return helper(k, n, memo);
    }
};
```

iv. Memoization + Binary Search, It will pass the Test Cases

O(nlogk)

```
class Solution {
public:
    //Time: O(n*k*logn), Space: O(n*k)
    int helper(int k, int n, vector<vector<int>>& memo){
        if(n == 0 || n == 1) return n;
        if(k == 1) return n;

        if(memo[k][n] != -1) return memo[k][n];

        int mn = INT_MAX, low = 0, high = n, temp = 0;

        while(low<=high){

            int mid = (low + high)/2;

            /*representing both the choices with memo
            First one, if the egg will break, no. of eggs will decreased and we have to
            down from that floor.
            Second one, if the egg will not break, no. of eggs will not decreased and we
            have to go above form that floor.*/

            int left = helper(k-1, mid-1, memo);
            int right = helper(k, n-mid, memo);

            temp = 1 + max(left, right);

            //since we need more temp value in worst case, so need to go above
            if(left < right)
                low = mid+1;
            else
                high = mid-1;     //move to the downward

            mn = min(mn, temp); //minimum number of attempts
        }
        return memo[k][n] = mn;
    }

    int superEggDrop(int k, int n) {
        //k means number of eggs, n means number of floors
        vector<vector<int>> memo(k+1, vector<int>(n+1, -1));
        return helper(k, n, memo);
    }
};
```

```
#define vi vector<int>
#define vvi vector<vi>
class Solution {
public:
    //Time: O(n*k*logn), Space: O(n*k)
    int f(int k, int n, vvi& dp){
        if(n == 0 or n == 1) return n;
        if(k==1) return n; //that is the base case.
        if(dp[k][n] != -1) return dp[k][n]; //this is filling the DP table.
        int mn = INT_MAX, low = 0, high = n, temp = 0; //this is vanilla binary search
        while(low<=high){ //standard binary search
            int mid = low + (high -low)/2; //this is mid to avoid overflow.
            /*representing both the choices with memo
            First one, if the egg will break, no. of eggs will decreased and we have to
            down from that floor.
            Second one, if the egg will not break, no. of eggs will not decreased and we
            have to go above form that floor.*/
            int left = f(k-1, mid-1, dp); //1 egg is broken and egg thrown from upper floors
            int right = f(k, n-mid, dp); // no egg is broken since egg is thrown from a lower floor.
            temp = 1 + max(left, right); //number of attempts will be 1+ max of left and right.
            //since we need more temp value in worst case, so need to go above
            if(left < right) low = mid+1; //move to the upper half side.
            else high = mid-1;     //move to the downward side.
            mn = min(mn, temp); //minimum number of attempts, and keep on updating it.
        }
        return dp[k][n] = mn; //assign the minimum number of eggs required for n eggs and if you are on kth floor.
    }
    int superEggDrop(int k, int n) {
        //k means number of eggs, n means number of floors
        vvi dp(k+1, vector<int>(n+1, -1)); //initialisation
        return f(k, n, dp); //this will return the minimum number of eggs required if thrown from a floor.
    }
};
```

887. Super Egg Drop

Hard 2094 115 Add to List Share

You are given `k` identical eggs and you have access to a building with `n` floors labeled from `1` to `n`.

You know that there exists a floor `f` where `0 <= f <= n` such that any egg dropped at a floor **higher** than `f` will **break**, and any egg dropped **at or below** floor `f` will **not break**.

Each move, you may take an unbroken egg and drop it from any floor `x` (where `1 <= x <= n`). If the egg breaks, you can no longer use it. However, if the egg does not break, you may **reuse** it in future moves.

Return *the **minimum number of moves** that you need to determine **with certainty** what the value of* `f` is.

Example 1:

```
Input: k = 1, n = 2
Output: 2
Explanation:
Drop the egg from floor 1. If it breaks,
we know that f = 0.
Otherwise, drop the egg from floor 2. If
it breaks, we know that f = 1.
If it does not break, then we know f = 2.
Hence, we need at minimum 2 moves to
determine with certainty what the value of
f is.
```

Consecutive 1's not allowed

Medium Accuracy: 48.5% Submissions: 15141
Points: 4

Given a positive integer **N**, count all possible distinct binary strings of length **N** such that there are **no consecutive 1's**. Output your answer **modulo 10^9 + 7.**

Example 1:

```
Input:
N = 3
Output: 5
Explanation: 5 strings are (000,
001, 010, 100, 101).
```

Example 2:

```
Input:
N = 2
Output: 3
Explanation: 3 strings are
(00,01,10).
```

Your Task:

F. A Random Code Problem

time limit per test: 3 seconds
memory limit per test: 512 megabytes
input: standard input
output: standard output

You are given an integer array $a_0, a_1, \ldots, a_{n-1}$, and an integer k. You perform the following code with it:

```
long long ans = 0; // create a 64-bit signed variable which is initially equal to 0
for(int i = 1; i <= k; i++)
{
  int idx = rnd.next(0, n - 1); // generate a random integer between 0 and n - 1, both inclusive
                                // each integer from 0 to n - 1 has the same probability of being chosen
  ans += a[idx];
  a[idx] -= (a[idx] % i);
}
```

Your task is to calculate the expected value of the variable `ans` after performing this code.

Note that the input is generated according to special rules (see the input format section).

Input

The only line contains six integers n, a_0, x, y, k and M ($1 \le n \le 10^7$; $1 \le a_0, x, y < M \le 998244353$; $1 \le k \le 17$).

The array a in the input is constructed as follows:

- a_0 is given in the input;
- for every i from 1 to $n-1$, the value of a_i can be calculated as $a_i = (a_{i-1} \cdot x + y) \bmod M$.

Output

Let the expected value of the variable `ans` after performing the code be E. It can be shown that $E \cdot n^k$ is an integer. You have to output this integer modulo 998244353.

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

using namespace std;

const int MOD = 998244353;
const int L = 720720;

int add(int x, int y, int m = MOD)
{
    x += y;
    if(x >= m) x -= m;
    return x;
}

int mul(int x, int y, int m = MOD)
{
    return (x * 1ll * y) % m;
}

int binpow(int x, int y)
{
    int z = 1;
    while(y > 0)
    {
        if(y % 2 == 1) z = mul(z, x);
        x = mul(x, x);
        y /= 2;
    }
    return z;
}

int inv(int x)
{
    return binpow(x, MOD - 2);
}

int divide(int x, int y)
{
    return mul(x, inv(y));
}

int main()
{
    int n, a0, x, y, k, M;
    cin >> n >> a0 >> x >> y >> k >> M;
    vector<int> arr(n);
    arr[0] = a0;
    for(int i = 1; i < n; i++)
        arr[i] = add(mul(arr[i - 1], x, M), y, M);
    int ans = 0;
    int total_ways = binpow(n, k);
    int coeff = mul(divide(total_ways, n), k);
    vector<vector<int>> dp(k, vector<int>(L));
    for(int i = 0; i < n; i++)
    {
        int p = arr[i] / L;
        int q = arr[i] % L;
        dp[0][q]++;
        ans = add(ans, mul(p, mul(L, coeff)));
    }
    int cur_coeff = divide(total_ways, n);
    for(int i = 1; i <= k; i++)
    {
        for(int j = 0; j < L; j++)
        {
            int cur = dp[i - 1][j];
            if(i < k)
                dp[i][j] = add(dp[i][j], mul(n - 1, cur));
            ans = add(ans, mul(j, mul(cur, cur_coeff)));
            if(i < k)
                dp[i][j - (j % i)] = add(dp[i][j - (j % i)], cur);
        }
        cur_coeff = divide(cur_coeff, n);
    }
    cout << ans << endl;
}
```

486. Predict the Winner

Medium 2592 136 Add to List Share

You are given an integer array `nums`. Two players are playing a game with this array: player 1 and player 2.

Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of `0`. At each turn, the player takes one of the numbers from either end of the array (i.e., `nums[0]` or `nums[nums.length - 1]`) which reduces the size of the array by `1`. The player adds the chosen number to their score. The game ends when there are no more elements in the array.

Return `true` if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return `true`. You may assume that both players are playing optimally.

Example 1:

```
Input: nums = [1,5,2]
Output: false
Explanation: Initially, player 1 can choose between 1 and 2.
If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2).
So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
Hence, player 1 will never be the winner and you need to return false.
```

Example 2:

```
Input: nums = [1,5,233,7]
Output: true
Explanation: Player 1 first chooses 1. Then player 2 has to choose between 5 and 7. No matter which number player 2 choose, player 1 can choose 233.
Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player1 can win.
```

Optimal Strategy For A Game

Medium Accuracy: 52.29% Submissions: 17821 Points: 4

You are given an array **A of size N**. The array contains integers and is of **even length**. The elements of the array represent N **coin** of **values $V_1, V_2, ...V_n$**. You play against an opponent in an **alternating** way.

In each **turn**, a player selects either the **first or last coin** from the **row**, removes it from the row permanently, and **receives the value** of the coin.

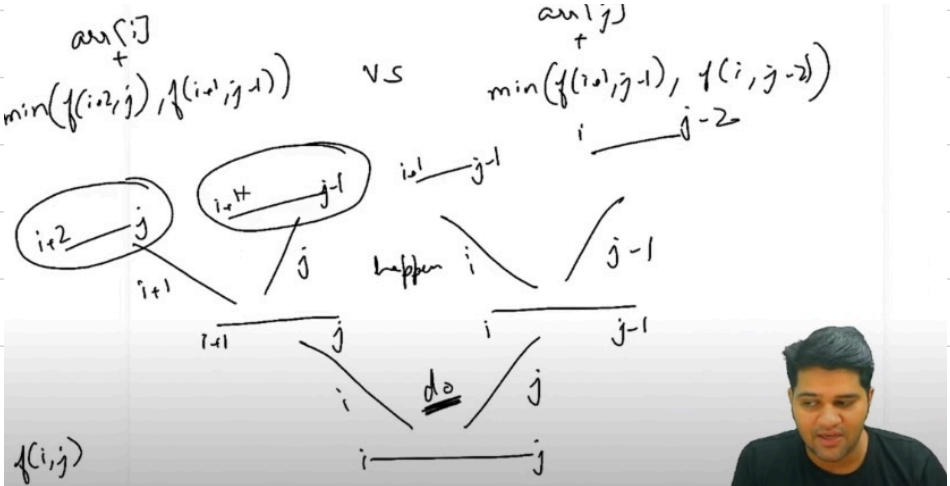
You need to determine the **maximum possible amount of money** you can win if you **go first**.
Note: Both the players are playing optimally.

Example 1:

```
Input:
N = 4
A[] = {5,3,7,10}
Output: 15
Explanation: The user collects maximum
value as 15(10 + 5)
```

```cpp
#define vi vector<int>
#define vvi vector<vi>
class Solution {
    int f(int i, int j, vi &P,vvi &dp) {
        if (dp[i][j]!=-1) return dp[i][j];
        if(i==j-1) return dp[i][j] = max(P[i], P[j]);
        auto a = P[i] + min(f(i+2,j,P,dp), f(i+1,j-1,P,dp));
        auto b = P[j] + min(f(i+1,j-1,P,dp), f(i,j-2,P,dp));
        return dp[i][j]=max(a,b);
    }
    public:
    bool stoneGame(vi &P) {
        int tot = 0;
        for (auto &x : P) tot += x;
        vvi dp(P.size(), vi(P.size(), -1));
        return f(0, P.size()-1, P, dp) > tot/2;
    }
};
```

```cpp
//Function to find the maximum possible amount of money we can win.
class Solution{
    public:
    long long int solve(long long int dp[][1001],int arr[],int n, int i, int j){

        if(i>j) return dp[i][j]=0;
        if(i==j) return dp[i][j]=arr[i];

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

        long long int ith_pick=arr[i]+ min(solve(dp,arr,n,i+2,j),solve(dp,arr,n,i+1,j-1));
        long long int jth_pick=arr[j]+ min(solve(dp,arr,n,i,j-2),solve(dp,arr,n,i+1,j-1));

        return dp[i][j]=max(ith_pick,jth_pick);
    }


    long long maximumAmount(int arr[], int n){

       long long int dp[1001][1001];
        memset(dp,-1,sizeof(dp));

        return solve(dp,arr,n,0,n-1);
    }
};
```



Palindromic patitioning

Hard Accuracy: 52.73% Submissions: 33889 Points: 8

Given a string **str**, a partitioning of the string is a *palindrome partitioning* if every sub-string of the partition is a palindrome. Determine the fewest cuts needed for palindrome partitioning of given string.

Example 1:

```
Input: str = "ababbbabbababa"
Output: 3
Explaination: After 3 partitioning substrings
are "a", "babbbab", "b", "ababa".
```

Example 2:

```
Input: str = "aaabba"
Output: 1
Explaination: The substrings after 1
partitioning are "aa" and "abba".
```

Your Task:
You do not need to read input or print anything, Your task is to complete the function **palindromicPartition()** which takes the string str as input parameter and returns minimum number of partitions required.

Expected Time Complexity: O(n*n) [n is the length of the string str]
Expected Auxiliary Space: O(n*n)

```
class Solution
{
    public:
    int dp[501][501];
    bool isPalindrome(string str, int i,int j){
        while(i<j){
            if(str[i]!=str[j])return false;
            i++;
            j--;
        }
        return true;
    }
    int f(string str,int i,int j){
        int left,right;
        if(i>=j or isPalindrome(str,i,j)==true) return 0;
        if(dp[i][j]!=-1)return dp[i][j];
        int ans=INT_MAX;
        for(int k=i;k<j;k++){
            if(dp[i][k]!=-1) left=dp[i][k];
            else{
                left=f(str,i,k);
                dp[i][k]=left;
            }
            if(dp[k+1][j]!=-1) right=dp[k+1][j];
            else{
                right=f(str,k+1,j);
                dp[k+1][j]=right;
            }
            int temp=left+right+1;
            ans = min(ans,temp);
        }
        return dp[i][j]=ans;
    }
    int palindromicPartition(string str){
        int n=str.length();
        memset(dp,-1,sizeof(dp));
        int i=0;
        int j=n-1;
        return f(str,i,j);
    }
```

WE will use DP to solve this problem.

This question is the variation of matrix chain multiplication.
We need to return minimum number of cuts so that the entire string Is partitioned into palindromes

If you are given with the string, and it is a palindrome itself then We have to return a 0 as the string itself is a palindrome so there are no cuts required.

This is the recursive approach to solve this question which is also the optimal substructure property.

Using Recursion

```
// i is the starting index and j is the ending index. i must be passed as 0 a
minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.
minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can
// calculated recursively using the following formula.
minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +
                                 minPalPartion(str, k+1, j) }
                           where k varies from i to j-1
```

So we are passing the original string, start index I and end injex j
So if I=j =0 then the function f(s,i,j) will yield 0.
Also f(s,i,j) will give 0 if s is itself a palindrome

These are the two base cases of the recursion. Rest we will give the magic Power to the recursion fairy and let her decide what she can do.

Now f(s,i,j) = min(f(s,i,k)+1+f(s,k+1,j) where k ={i,j-1}

```
bool isPalindrome(string String, int i, int j)
{
    while(i < j)
    {
      if(String[i] != String[j])
        return false;
      i++;
      j--;
    }
    return true;
}
int minPalPartion(string String, int i, int j)
{
    if( i >= j || isPalindrome(String, i, j) )
      return 0;
    int ans = INT_MAX, count;
    for(int k = i; k < j; k++)
    {
      count = minPalPartion(String, i, k) +
        minPalPartion(String, k + 1, j) + 1;

      ans = min(ans, count);
    }
    return ans;
```

So we are checking that if the string s is palindrome or not, from the index i to index j then we write a while loop which runs from I to J until I surpasses J it keeps on running.

We check if the character at index is not equal to character at index J then we return false, otherwise we keep on incrementing I and decrementing J and at the end, if no-one of the instances return false then at the end, we return true, this means the substring from I to J is a palindrome.

The function minPalPartition takes three inputs String s, start index I and end index J.

We check if I>=j and if the string from I to j is palindrome if anyone of these cases happen we return true, this means if the I has surpasses or equal to j or the string is a palindrome string itself, we need to return true as these were the base cases.

Now we store the number of bars in answer. As we need to return minimum such bars so we assign it to INT_MAX in the beginning.

And we store a variable count.

Now, we run a for loop from k=I to j-1 and store the count in count

Count = f(s,i,k) + f(s,k+1,j) + 1
and in every iteration check the min of both bars and count in bars. And return bars at the end.

But this recursive calls are repetitive in nature. So we will make use of DP and store the result in DP memo

Top down approach

```
// Dynamic Programming Solution for
// Palindrome Partitioning Problem
#include <bits/stdc++.h>
using namespace std;

// Returns the minimum number of cuts
// needed to partition a string
// such that every part is a palindrome
int minPalPartion(string str)
{
    // Get the length of the string
    int n = str.length();

    /* Create two arrays to build the solution
       in bottom up manner
    C[i][j] = Minimum number of cuts needed for
              palindrome partitioning
              of substring str[i..j]
    P[i][j] = true if substring str[i..j] is
              palindrome, else false
    Note that C[i][j] is 0 if P[i][j] is true */
    int C[n][n];
    bool P[n][n];

    // Every substring of length 1 is a palindrome
    for (int i = 0; i < n; i++) {
        P[i][i] = true;
        C[i][i] = 0;
    }

    for (int L = 2; L <= n; L++) {

        // For substring of length L, set
        // different possible starting indexes
        for (int i = 0; i < n - L + 1; i++) {
            int j = i + L - 1; // Set ending index

            // If L is 2, then we just need to
            // compare two characters. Else
            // need to check two corner characters
            // and value of P[i+1][j-1]
            if (L == 2)
                P[i][j] = (str[i] == str[j]);
            else
                P[i][j] = (str[i] == str[j]) && P[i + 1][j - 1];

            // IF str[i..j] is palindrome, then C[i][j] is 0
            if (P[i][j] == true)
                C[i][j] = 0;
            else {

                // Make a cut at every possible
                // location starting from i to j,
                // and get the minimum cost cut.
                C[i][j] = INT_MAX;
                for (int k = i; k <= j - 1; k++)
                    C[i][j] = min(C[i][j], C[i][k] + C[k + 1][j] + 1);
            }
        }
    }
    // Return the min cut value for
    // complete string. i.e., str[0..n-1]
    return C[0][n - 1];
}
```

This is the further optimised code for the above question, so We can optimize the above code a bit further. Instead of calculating C[i] separately in O(n^2), we can do it with the P[i] itself.

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

int minCut(string a)
{
    int cut[a.length()];
    bool palindrome[a.length()][a.length()];
    memset(palindrome, false, sizeof(palindrome));
    for (int i = 0; i < a.length(); i++)
    {
        int minCut = i;
        for (int j = 0; j <= i; j++)
        {
            if (a[i] == a[j] && (i - j < 2 || palindrome[j + 1][i - 1]))
            {
                palindrome[j][i] = true;
                minCut = min(minCut, j == 0 ? 0 : (cut[j - 1] + 1));
            }
        }
        cut[i] = minCut;
    }
    return cut[a.length() - 1];
}
```

Time Complexity: $O(n^2)$

Using Memorization to solve this problem.

The basic idea is to cache the intermittent results calculated in recursive functions. We can put these results into a hashmap/unordered_map.

To calculate the keys for the Hashmap we will use the starting and end index of the string as the key i.e. ["start_index".append("end_index")] would be the key for the Hashmap.

Below is the implementation of above approach :

```
// Using memoizatoin to solve the partition problem.
#include <bits/stdc++.h>
using namespace std;
// Function to check if input string is palindrome or not
bool is palindrome(string input, int start, int end)
{
    // Using two pointer technique to check palindrome
    while (start < end) {
        if (input[start] != input[end])
            return false;
        start++;
        end--;
    }
    return true;
}

// Function to find keys for the Hashmap
string convert(int a, int b)
{
    return to_string(a) + "" + to_string(b);
}
```

Imagine you have a special keyboard with the following keys:

Key 1: Prints 'A' on screen
Key 2: (Ctrl-A): Select screen
Key 3: (Ctrl-C): Copy selection to buffer
Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Find maximum numbers of A's that can be produced by pressing keys on the special keyboard N times.

Example 1:

```
Input: N = 3
Output: 3
Explaination: Press key 1 three times.
```

Example 2:

```
Input: N = 7
Output: 9
Explaination: The best key sequence is
key 1, key 1, key 1, key 2, key 3,
key4, key 4.
```

Your Task:
You do not need to read input or print anything. Your task is to complete the function **optimalKeys()** which takes N as input parameter and returns the maximum number of A's that can be on the screen after performing N operations.

Expected Time Complexity: $O(N^2)$
Expected Auxiliary Space: O(N)

shashankrustagii 2 minutes ago

```
//https://www.youtube.com/watch?v=c_y7H7qZJRU
class Solution{
public:
    long long int optimalKeys(int N)
    {
        if(N <=6) return N;
        vector<int> dp(N+1,0);
        for(int i=1;i<=6;i++) dp[i]=i;
        for(int i=7;i<=N;i++){
            for(int j=i-3;j>=1;j--){
                int curr = dp[j]*(i-j-1);
                dp[i]=max(curr,dp[i]);
            }

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

Perfect Sum Problem

Medium Accuracy: 47.14% Submissions: 25980 Points: 4

Given an array **arr[]** of integers and an integer **sum**, the task is to count all subsets of the given array with a sum equal to a given **sum**.

Note: Answer can be very large, so, output answer modulo 10^9+7

Example 1:

```
Input: N = 6, arr[] = {2, 3, 5, 6, 8, 10}
       sum = 10
Output: 3
Explanation: {2, 3, 5}, {2, 8}, {10}
```

Example 2:

```
Input: N = 5, arr[] = {1, 2, 3, 4, 5}
       sum = 10
Output: 3
Explanation: {1, 2, 3, 4}, {1, 4, 5},
             {2, 3, 5}
```

Your Task:
You don't need to read input or print anything. Complete the function **perfectSum ()** which takes **N,** array **arr[]** and **sum** as input parameters and returns an integer value

Expected Time Complexity: O(**N*sum**)
Expected Auxiliary Space: O(**N*sum**)

Constraints:
$1 \le N*sum \le 10^6$

```
class Solution{

    public:
    int perfectSum(int arr[], int n, int sum)
    {
        long long dp[n+1][sum+1];
        long long mod=1e9+7;
        int zero=0;
        for(int i=0;i<n;i++) if(arr[i]==0)zero++;
        for(int i=0;i<=n;i++) dp[i][0]=1;
        for(int i=1;i<=sum;i++)dp[0][i]=0;
        for(int i=1;i<=n;i++)
            for(int j=1;j<=sum;j++)
                if(arr[i-1]<=j && arr[i-1])
                    dp[i][j]=((dp[i-1][j]%mod)+(dp[i-1][j-arr[i-1]]%mod))%mod;
                else dp[i][j]=dp[i-1][j]%mod;

        return pow(2,zero)*dp[n][sum];
    }
};
```

Longest Bitonic subsequence

Medium Accuracy: 49.11% Submissions: 8480 Points: 4

Given an array of positive integers. Find the maximum length of Bitonic subsequence. A subsequence of array is called Bitonic if it is first increasing, then decreasing.

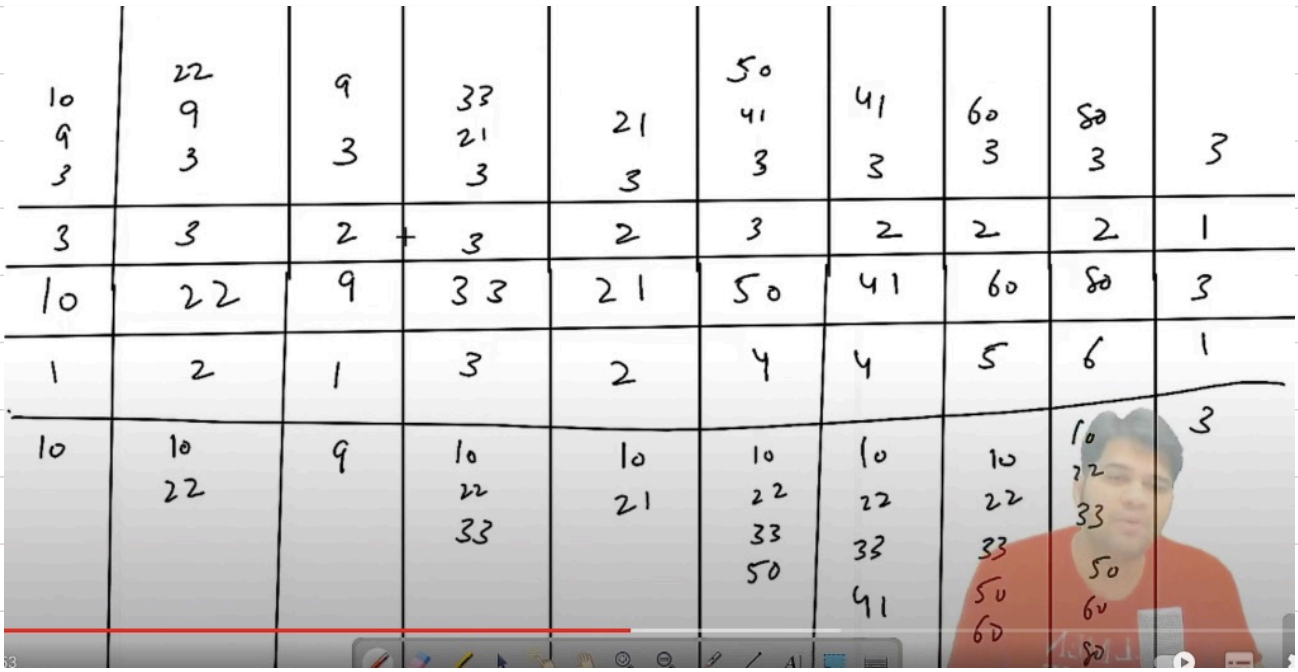
Example 1:

```
Input: nums = [1, 2, 5, 3, 2]
Output: 5
Explanation: The sequence {1, 2, 5} is
increasing and the sequence {3, 2} is
decreasing so merging both we will get
length 5.
```

Example 2:

```
Input: nums = [1, 11, 2, 10, 4, 5, 2, 1]
Output: 6
Explanation: The bitonic sequence
{1, 2, 10, 4, 2, 1} has length 6.
```

```cpp
class Solution{
    public:
    int LongestBitonicSequence(vector<int>nums)
    {
        int n = nums.size();
        int forward[n], backward[n];
        forward[0] = 1;
        for(int i=1; i<n; i++) {
            forward[i] = 1;
            for(int j=i-1; j>=0; j--) {
                if(nums[j]>=nums[i]) continue;
                forward[i] = max(forward[j]+1, forward[i]);
            }
        }
        backward[n-1] = 1;
        for(int i = n-2; i>=0; i--) {
            backward[i] = 1;
            for(int j=i+1; j<n; j++) {
                if(nums[j]>=nums[i]) continue;
                backward[i] = max(backward[j]+1, backward[i]);
            }
        }
        int ans=INT_MIN;
        for(int i=0; i<n; i++) {
            ans = max(ans, forward[i]+backward[i]-1);
        }
        return ans;
    }
};
```



This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. Let the input array be arr[] of length n. We need to construct two arrays lis[] and lds[] using Dynamic Programming solution of LIS problem. lis[i] stores the length of the Longest Increasing subsequence ending with arr[i]. lds[i] stores the length of the longest Decreasing subsequence starting from arr[i]. Finally, we need to return the max value of lis[i] + lds[i] – 1 where i is from 0 to n-1. Following is the implementation of the above Dynamic Programming solution.

Time Complexity: O(n^2)
Auxiliary Space: O(n)

Longest Common Substring

Medium Accuracy: 52.09% Submissions: 37396 Points: 4

Given two strings. The task is to find the length of the longest common substring.

Example 1:

```
Input: S1 = "ABCDGH", S2 = "ACDGHR"
Output: 4
Explanation: The longest common substring
is "CDGH" which has length 4.
```

Example 2:

```
Input: S1 = "ABC", S2 "ACB"
Output: 1
Explanation: The longest common substrings
are "A", "B", "C" all having length 1.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **longestCommonSubstr()** which takes the string S1, string S2 and their length n and m as inputs and returns the length of the longest common substring in S1 and S2.

Expected Time Complexity: O(n*m).
Expected Auxiliary Space: O(n*m).

Constraints:
1<=n, m<=1000

```cpp
class Solution{
    public:

    int longestCommonSubstr (string s1, string s2, int n, int m)
    {
        int dp[n+1][m+1];
        int maxlength=0;
        for(int i=0;i<=n;i++) dp[i][0] = 0;
        for(int i=0;i<=m;i++) dp[0][i] = 0;

        for(int i=1;i<=n;i++)
            for(int j=1;j<=m;j++)
            {
                if(s1[i-1]==s2[j-1]) dp[i][j] = 1 + dp[i-1][j-1];
                else dp[i][j]=0;
                maxlength = max(maxlength, dp[i][j]);
            }
        return maxlength;
    }
};
```

Mobile numeric keypad

Medium Accuracy: 49.98% Submissions: 10755 Points: 4

Given the mobile numeric keypad. You can only press buttons that are up, left, right, or down to the current button. You are not allowed to press bottom row corner buttons (i.e. * and #). Given a number **N**, the task is to find out the number of possible numbers of the given length.

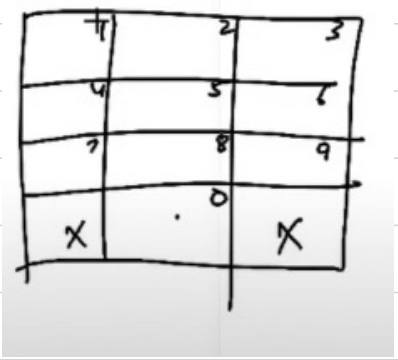
Example 1:

```
Input: 1
Output: 10
Explanation: Number of possible numbers
would be 10 (0, 1, 2, 3, …., 9)
```

Example 2:

```
Input: N = 2
Output: 36
Explanation: Possible numbers: 00, 08, 11,
12, 14, 22, 21, 23, 25 and so on.
If we start with 0, valid numbers
will be 00, 08 (count: 2)
If we start with 1, valid numbers
will be 11, 12, 14 (count: 3)
If we start with 2, valid numbers
will be 22, 21, 23,25 (count: 4)
If we start with 3, valid numbers
will be 33, 32, 36 (count: 3)
If we start with 4, valid numbers
will be 44,41,45,47 (count: 4)
If we start with 5, valid numbers
will be 55,54,52,56,58 (count: 5)
and so on..
```

```cpp
class Solution{
    public:
    long long getCount(int N){
        vector<int> choice[10]={{0,8},{1,2,4},{1,2,3,5},
        {2,3,6},{1,4,5,7},{2,4,5,6,8},
        {3,5,6,9},{4,7,8},{5,7,8,9,0}
        ,{6,8,9}};
        vector<long long> dp(11);
        for(int i=0;i<=9;i++) dp[i]=1;
        long long sum=0;
        for(int i=2;i<=N;i++){
            vector<long long> dp1(11);
            for(int num = 0;num<= 9;num++)
                for(auto a:choice[num]) dp1[a]+=dp[num];
            dp=dp1;
        }
        for(int i=0;i<=9;i++) sum+=dp[i];
        return sum;
    }
};
```



Maximum Profit

Hard Accuracy: 61.29% Submissions: 9931 Points: 8

In the stock market, a person buys a stock and sells it on some future date. Given the stock prices of **N** days in an array **A[ ]** and a positive integer **K**, find out the maximum profit a person can make in at-most **K** transactions. A transaction is equivalent to (buying + selling) of a stock and new transaction can start only when the previous transaction has been completed.

Time and space: O(nK)

Example 1:

```
Input: K = 2, N = 6
A = {10, 22, 5, 75, 65, 80}
Output: 87
Explaination:
1st transaction: buy at 10 and sell at 22.
2nd transaction : buy at 5 and sell at 80.
```

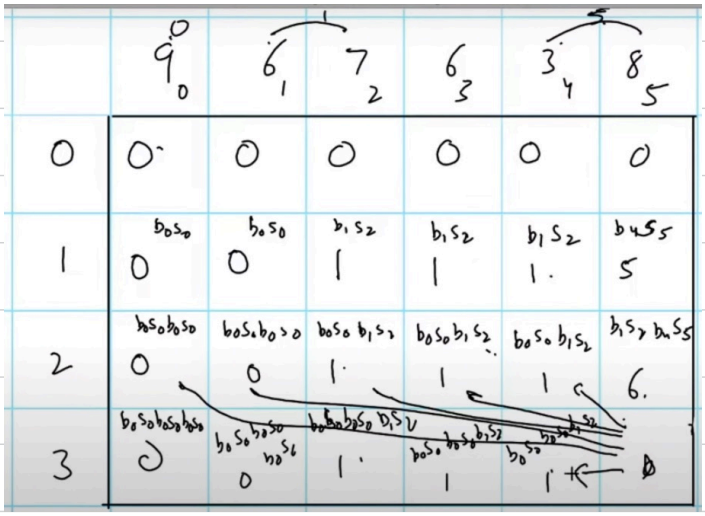
Example 2:

```
Input: K = 3, N = 4
A = {20, 580, 420, 900}
Output: 1040
Explaination: The trader can make at most 2
transactions and giving him a profit of 1040.
```

Example 3:

```
Input: K = 1, N = 5
A = {100, 90, 80, 50, 25}
Output: 0
Explaination: Selling price is decreasing
daily. So seller cannot have profit.
```

```
class Solution {
  public:
    int dp[501][201][2];
    int helper(int prices[], int curr, int n, int k, bool hadBought){
        if(curr>=n){
            return 0;
        }
        if(k==0){
            return 0;
        }
        int profit=0;
        if(dp[curr][k][hadBought]!=-1){
            return dp[curr][k][hadBought];
        }
        if(hadBought==false){
            int BUY=(-prices[curr])+helper(prices,curr+1,n,k,true);
            //think as money going out of pocket
            int NOT_BUY=helper(prices,curr+1,n,k,hadBought);
            //do not buy / ignore
            profit=max(BUY,NOT_BUY);
        }
        else{
            int SELL=(prices[curr])+helper(prices,curr+1,n,k-1,false);
            //think as money coming in our pocket
            int NOT_SELL=helper(prices,curr+1,n,k,hadBought);
            //ignore
            profit=max(SELL,NOT_SELL);
        }
        return dp[curr][k][hadBought]=profit;
    }
    int maxProfit(int k, int n, int A[]) {
        memset(dp,-1,sizeof(dp));
        return helper(A,0,n,k,false);
    }

};
```



```
class Solution {
  public:
    int maxProfit(int k, int n, int a[]) {
        int dp[k+1][n];
        memset(dp,0,sizeof(dp));
        for(int i=1;i<=k;i++)
        {
            int mx=INT_MIN;
            for(int j=1;j<n;j++)
            {
                mx=max(mx,dp[i-1][j-1]-a[j-1]);
                dp[i][j]=max(dp[i][j-1],mx+a[j]);
            }
        }
        return dp[k][n-1];
    }
};
```

Time: O(kn)
Space: O(n)

```
int[] arr = new int[n];
for(int i = 0; i < arr.length; i++){
    arr[i] = scn.nextInt();
}

int k = scn.nextInt();

int[][] dp = new int[k + 1][n];

for(int t = 1; t <= k; t++){
    int max = Integer.MIN_VALUE;

    for(int d = 1; d < arr.length; d++){
        if(dp[t - 1][d - 1] - arr[d - 1] > max){
            max = dp[t - 1][d - 1] - arr[d - 1];
        }

        if(max + arr[d] > dp[t][d - 1]){
            dp[t][d] = max + arr[d];
        } else {
            dp[t][d] = dp[t][d - 1];
        }
    }
}
```

GFG HARD

Longest arithmetic Progression

Longest Arithmetic Progression

Medium Accuracy: 44.16% Submissions: 4309 Points: 4

Given an array called **A[]** of sorted integers having no duplicates, find the length of the **L**ongest **A**rithmetic **P**rogression (LLAP) in it.

Example 1:

```
Input:
N = 6
set[] = {1, 7, 10, 13, 14, 19}
Output: 4
Explanation: The longest arithmetic
progression is {1, 7, 13, 19}.
```

Example 2:

```
Input:
N = 5
A[] = {2, 4, 6, 8, 10}
Output: 5
Explanation: The whole set is in AP.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **lenghtOfLongestAP()** which takes the array of integers called **set[]** and **n** as input parameters and returns the length of LLAP.

Expected Time Complexity: $O(N^2)$
Expected Auxiliary Space: $O(N^2)$

```cpp
class Solution{
public:
    int lengthOfLongestAP(int arr[], int n) {
        if(n==1) return 1;

       if(n==2) return 2;

       vector<unordered_map<int,int>> dp(n);
       int ans = 0;
       for(int i=1;i<n;i++)
       {

           for(int j=0;j<i;j++)
           {
               int diff = arr[i] - arr[j];

               if(dp[j].find(diff)==dp[j].end())
               {
                   dp[i][diff] = 2;
               }else
               {
                   dp[i][diff] = dp[j][diff]+1;
               }
               ans = max(ans,dp[i][diff]);

           }
       }

       return ans;
    }
};
```

Minimum Squares

```cpp
class Solution{
public:

int lengthOfLongestAP(int nums[], int n) {
        int ans = 0;
        if(n == 1) return 1;

        vector<vector<int>>dp(n, vector<int>(10001));

        for(int i = 0; i < n; i++) {
            for(int j = i + 1; j < n; j++) {

                int d = nums[j] - nums[i];

                dp[j][d] = max(dp[i][d] + 1, 2);

                ans = max(ans, dp[j][d]);
            }
        }

        return ans;
    }
};
```

```cpp
class Solution
{
    public:
    int lengthOfLongestAP(int nums[], int n)
    {
        int ans = 0;
        if(n == 1) return 1;
        vector<vector<int>>dp(n, vector<int>(10001));
        for(int i = 0; i < n; i++)
            for(int j = i + 1; j < n; j++)
            {
                int d = nums[j] - nums[i];
                ans = max(ans, dp[j][d]=max(dp[i][d]+1,2));
            }
        return ans;
    }
};
```

Get Minimum Squares

Medium Accuracy: 59.96% Submissions: 8145 Points: 4

Given a number N. Find the minimum number of squares of any number that sums to N. For Example: If N = 100 , N can be expressed as (10*10) and also as (5*5 + 5*5 + 5*5 + 5*5) but the output will be 1 as minimum number of square is 1 , i.e (10*10).

Example 1:

```
Input: N = 100
Output: 1
Explanation: 10 * 10 = 100
```

Example 2:

```
Input: N = 6
Output: 3
Explanation = 1 * 1 + 1 * 1 + 2 * 2 = 6
```

```cpp
class Solution{
    public:
    int MinSquares(int n)
    {
        int dp[n+1];
        int a = sqrt(n);
        dp[0] = 0, dp[1] = 1, dp[2] = 2, dp[3] = 3;
        for(int i=4; i<=n; i++) {
            dp[i] = i;
            for(int j=1; j*j<=i; j++) {
                dp[i] = min(dp[i], 1+dp[i-j*j]);
            }
        }
        return dp[n];
    }
};
```

A number can always be represented as a sum of squares of other numbers. Note that 1 is a square and we can always break a number as (1*1 + 1*1 + 1*1 + ...). Given a number n, find the minimum number of squares that sum to X.

The idea is simple, we start from 1 and go to a number whose square is smaller than or equals n. For every number x, we recur for n-x. Below is the recursive formula.

```
int getMinSquares(unsigned int n)
{
    // base cases
    // if n is perfect square then
    // Minimum squares required is 1
    // (144 = 12^2)
    if (sqrt(n) - floor(sqrt(n)) == 0)
        return 1;
    if (n <= 3)
        return n;

    // getMinSquares rest of the
    // table using recursive
    // formula
    // Maximum squares required
    // is n (1*1 + 1*1 + ..)
    int res = n;

    // Go through all smaller numbers
    // to recursively find minimum
    for (int x = 1; x <= n; x++)
    {
        int temp = x * x;
        if (temp > n)
            break;
        else
            res = min(res, 1 + getMinSquares
                                (n - temp));
    }
    return res;
}
```

The time complexity of the above solution is exponential. If we draw the recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from n = 6, we can reach 4 by subtracting one 2 times and by subtracting 2 one times. So the subproblem for 4 is called twice.

Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So min square sum problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array table[][] in a bottom-up manner. Below is a Dynamic programming-based solution.

We will create a function f

```
f(int n)
{
    if(n<=3) return n;
    int* dp = new int[n + 1];
    for(int I=0;i<=3;i++) dp[i]=i;
    for(int I=43;i<=n;i++)
    {
        dp[i]=i;
        for(int x=1;x<ceil(sqrt(x));x++)
        {
            if(x*x>I) break;
            else    dp[I]=min(dp[I],1+dp[I-x*x]);
        }
    }
    return dp[n];
}

}
```

BFS approach to solve this problem Time complexity: O(n*sqrt(n))

```
Create a visited vector of size n+1 of int type preinitialised to 0
vector<int> visited(n + 1,0);
Create a queue<pair<int,int>> q;
q.push({n,0});
int ans = INT_MAX;
visited[n]=1;
while(!q.empty())
{
    pair<int,int> p;
    p = q.front();
    q.pop();
    //If node reaches its destination 0, update the ans
    if(p.first==0) ans = max(ans,p.second)
    int path=(p.first - (i*i));
    if(path >= 0 && ( !visited[path]  or path ==0))
    {
      // Mark visited
      visited[path]=1;
      // Push it it Queue
      q.push({path,p.second + 1});
    }

}
Return ans;
```

The time complexity of the above problem is O(n)*sqrt(n) which is better than the Recursive approach.
Also, it is a great way to understand how BFS (Breadth-First Search) works.

Please write a if you find anything incorrect, or you want to share more information about the topic discussed above.

Another Approach:

This problem can also be solved using Dynamic programming (Bottom-up approach). The idea is like coin change 2 (in which we need to tell minimum number of coins to make an amount from given coins[] array), here an array of all perfect squares less than or equal to n can be replaced by coins[] array and amount can be replaced by n. Just see this as an unbounded knapsack problem, Let's see an example:

For given input n = 6, we will make an array upto 4, arr : [1,4]

Here, answer will be (4 + 1 + 1 = 6) i.e. 3.

Below is the implementation of the above approach:

minimum cost path, dungeon game leetcode

Minimum Points To Reach Destination

Hard Accuracy: 51.15% Submissions: 2757 Points: 8

Given a grid of size **M*N** with each cell consisting of an integer which represents points. We can move across a cell only if we have positive points. Whenever we pass through a cell, points in that cell are added to our overall points, the task is to find minimum initial points to reach cell (m-1, n-1) from (0, 0) by following these certain set of rules :

1. From a cell (i, j) we can move to (i + 1, j) or (i, j + 1).
2. We cannot move from (i, j) if your overall points at (i, j) are <= 0.
3. We have to reach at (n-1, m-1) with minimum positive points i.e., > 0.

Example 1:

```
Input: M = 3, N = 3
       arr[][] = {{-2,-3,3},
        {-5,-10,1},
        {10,30,-5}};

Output: 7
Explanation: 7 is the minimum value to
reach the destination with positive
throughout the path. Below is the path.
(0,0) -> (0,1) -> (0,2) -> (1, 2) -> (2, 2)
We start from (0, 0) with 7, we reach
(0, 1) with 5, (0, 2) with 2, (1, 2)
with 5, (2, 2) with and finally we have
1 point (we needed greater than 0 points
at the end).
```

```cpp
class Solution{

    public:
    int minPoints(vector<vector<int>> points, int M, int N)
    {
        int dp[M][N];
        dp[M-1][N-1] = 0;
        if(points[M-1][N-1]<0) dp[M-1][N-1] = points[M-1][N-1];
        for(int i=M-1; i>=0; i--)
        {
            for(int j=N-1; j>=0; j--)
            {
                if(i==M-1 && j==N-1) continue;
                if(i+1<M && j+1<N)
                {
                    dp[i][j] = points[i][j] + max(dp[i+1][j], dp[i][j+1]);
                    if(dp[i][j]>0)
                        dp[i][j] = 0;
                }
                else if(i+1<M)
                {
                    dp[i][j] = points[i][j] + dp[i+1][j];
                    if(dp[i][j]>0)
                        dp[i][j] = 0;
                }
                else if(j+1<N)
                {
                    dp[i][j] = points[i][j] + dp[i][j+1];
                    if(dp[i][j]>0)
                        dp[i][j] = 0;
                }
            }
        }
        return abs(dp[0][0])+1;
    }
};
```

174. Dungeon Game

Hard 3696 72 Add to List Share

The demons had captured the princess and imprisoned her in **the bottom-right corner** of a `dungeon`. The `dungeon` consists of `m x n` rooms laid out in a 2D grid. Our valiant knight was initially positioned in **the top-left room** and must fight his way through `dungeon` to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to `0` or below, he dies immediately.

Some of the rooms are guarded by demons (represented by negative integers), so the knight loses health upon entering these rooms; other rooms are either empty (represented as 0) or contain magic orbs that increase the knight's health (represented by positive integers).

To reach the princess as quickly as possible, the knight decides to move only **rightward** or **downward** in each step.

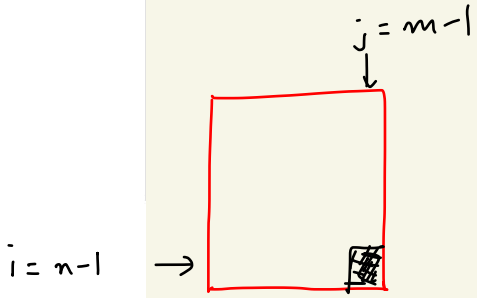
Return *the knight's minimum initial health so that he can rescue the princess*.

Note that any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Example 1:

-2 →	-3 →	3 ↓
-5	-10	1 ↓
10	30	-5

```
Input: dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]
Output: 7
Explanation: The initial health of the knight must be at least 7 if he follows the optimal path: RIGHT-> RIGHT -> DOWN -> DOWN.
```




```cpp
#define vi vector<int>
#define vvi vector<vi>
class Solution {
    long f(int i, int j, vvi &G,vvi &dp){
        if(i >= G.size() or j >= G[0].size()) return INT_MAX;
        if(i==G.size()-1 and j==G[0].size()-1){
            if(G[i][j]<=0) return abs(G[i][j])+1;
            else return 1;
        }
        if(dp[i][j]!=-1) return dp[i][j];
        auto down = f(i+1,j,G,dp);
        auto right = f(i,j+1,G,dp);
        int x = min(down, right)-G[i][j];
        if(x<=0) dp[i][j] = 1;
        else dp[i][j] = x;
        return dp[i][j];
    }
    public:
    int calculateMinimumHP(vvi &G) {
        vvi dp(205,vi(205,-1));
        return f(0, 0,G,dp);
    }
};
```

base case when either i or j has crossed the #r or #c

return stored

going ↓

going →

when we reached last cell

if -ve health, return 1+|h|

else return 1.

Top down dp.

Initialize the 2D MeMo

call it

Totally Decoding a message

Total Decoding Messages

Medium Accuracy: 43.69% Submissions: 25371 Points: 4

A top secret message containing letters from A-Z is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

You are an FBI agent. You have to determine the total number of ways that message can be decoded, as the answer can be large return the answer modulo $10^9 + 7$.
Note: An empty digit sequence is considered to have one decoding. It may be assumed that the input contains valid digits from 0 to 9 and If there are leading 0's, extra trailing 0's and two or more consecutive 0's then it is an invalid string.

Example 1:

```
Input: str = "123"
Output: 3
Explanation: "123" can be decoded as "ABC"(123),
"LC"(12 3) and "AW"(1 23).
```

Example 2:

```
Input: str = "27"
Output: 1
Explanation: "27" can be decoded as "BG".
```

Your Task:
You don't need to read or print anything. Your task is to complete the function **CountWays()** which takes the string as str as input parameter and returns the total number of ways the string can be decoded modulo $10^9 + 7$.

Expected Time Complexity: O(n)
Expected Space Complexity: O(n) where n = |str|

```
class Solution {
    public:
        int CountWays(string str){
            // Code here
            int n = str.size();
            const int mod = 1e9 + 7;
            if(str[0] == '0') return 0;
            for(int i = 0; i < n-1; i++)
                if(str[i] == '0' and str[i+1] == '0') return 0;
            int dp[n+1] = {0};
            dp[0] = 1;
            for(int i = 0; i < n; i++)
            {
                if(str[i] == '0') continue;
                dp[i+1] = (dp[i+1] + dp[i])%mod;
                if(i != n-1 and str[i] == '1' or (str[i] == '2' and str[i+1] < '7'))
                    dp[i+2] = (dp[i+2] + dp[i])%mod;
            }
            return dp[n];
        }
};
```

Boolean Paranthesisation

Boolean Parenthesization

Hard Accuracy: 49.75% Submissions: 27682 Points: 8

Given a boolean expression **S** of length **N** with following symbols.
Symbols
'T' ---> true
'F' ---> false
and following operators filled between symbols
Operators
& ---> boolean AND
| ---> boolean OR
^ ---> boolean XOR
Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Example 1:

```
Input: N = 7
S = T|T&F^T
Output: 4
Explaination: The expression evaluates
to true in 4 ways ((T|T)&(F^T)),
(T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T)).
```

Example 2:

```
Input: N = 5
S = T^F|F
Output: 2
Explaination: ((T^F)|F) and (T^(F|F)) are the
only ways.
```

We created an unordered map of string key and integer val
And then we used a helper function to recursively compute answers and update. F has parameters string, start index, End index and truth value.
Base case: if I exceeds J then return 0, if I is same as j and The truth value is true return true else return false and before returning store it in S[I] and then return . (Topdown)

Now I<j , for this, check in hash map if the hash map does not contain I J T or I J F then in that case store it in Key and insert that value m[key] at index key.

Expected Time Complexity: $O(N^3)$
Expected Auxiliary Space: $O(N^2)$

```
class Solution
{
    public:
    unordered_map<string,int> m;
    int mod = 1003;
    int F(string s,int i,int j,bool isTrue){
        //base cases
        if(i>j) return 0; //or return false
        if(i==j)
            if(isTrue==true)return s[i]=='T';
            else return s[i]=='F';
        //cheking in the hashmap
        string key = to_string(i)+" "+to_string(j)+" "+to_string(isTrue);
        if(m.count(key)!=0) return m[key];
        //recursive case
        int ans=0;
        for(int k=i+1;k<j;k=k+2)
        {
            int lt= F(s,i,k-1,true);
            int lf= F(s,i,k-1,false);
            int rt= F(s,k+1,j,true);
            int rf= F(s,k+1,j,false);
            //xor operator
            if(s[k]=='^')
                if(isTrue==true) ans += (lt*rf + lf*rt)%mod;
                else ans += (lt*rt + lf*rf)%mod;
            //or operator
            else if(s[k]=='|')
                if(isTrue==true) ans+= (lt*rt + lt*rf + lf*rt)%mod;
                else ans+= (lf*rf)%mod;
            //and operator
            else
                if(isTrue==true) ans+= (lt*rt)%mod;
                else ans+= (lf*rf + lt*rf + lf*rt)%mod;
        }
        return m[key]=ans%mod;
    }
    int countWays(int n, string s)
    {
        return F(s,0,n-1,true)%mod;
    }
};
```

Rest handle all the cases by recursion

Given a boolean expression with following symbols.

```
Symbols
    'T' ---> true
    'F' ---> false
```

And following operators filled between symbols

```
Operators
    &   ---> boolean AND
    |   ---> boolean OR
    ^   ---> boolean XOR
```

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.
Let the input be in form of two arrays one contains the symbols (T and F) in order and the other contains operators (&, | and ^}

Solution:
Let **T(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

Let **F(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to false.

Base Cases:

```
T(i, i) = 1 if symbol[i] = 'T'
T(i, i) = 0 if symbol[i] = 'F'

F(i, i) = 1 if symbol[i] = 'F'
F(i, i) = 0 if symbol[i] = 'T'
```

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ Total(i,k) * Total(k+1,j) - F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * F(k+1,j) + F(i,k) * T(k+1,j) & \text{If operator[k] is '\^'} \end{cases}$$

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

$$F(i,j) = \sum_{k=i}^{j-1} \begin{cases} Total(i,k) * Total(k+1,j) - T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * T(k+1,j) + F(i,k) * F(k+1,j) & \text{If operator[k] is '\^'} \end{cases}$$

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

Strictly Increasing Array

Hard Accuracy: 32.18% Submissions: 2271 Points: 8

Given an array **nums[]** of **N** positive integers. Find the minimum number of operations required to modify the array such that array elements are in strictly increasing order (A[i] < A[i+1]).
Changing a number to greater or lesser than original number is counted as one operation.

Example 1:

```
Input: nums[] = [1, 2, 3, 6, 5, 4]
Output: 2
Explanation: By decreasing 6 by 2 and
increasing 4 by 2, arr will be like
[1, 2, 3, 4, 5, 6] which is stricly
increasing.
```

Example 2:

```
Input: nums[] = [1, 2, 3, 4]
Output: 0
Explanation: Arrays is already strictly
increasing.
```

Your Task:
You don't need to read or print anything. Your task is to complete the function **min_opeartions()** which takes the array **nums[]** as input parameter and returns the minimum number of opeartion needed to make the array strictly increasing.

Expected Time Complexity: O(n^2)
Expected Space Complexity: O(n)

The problem is variation of Longest Increasing Subsequence. The numbers which are already a part of LIS need not to be changed. So minimum elements to change is difference of size of array and number of elements in LIS. Note that we also need to make sure that the numbers are integers. So while making LIS, we do not consider those elements as part of LIS that cannot form strictly increasing by inserting elements in middle.
Example { 1, 2, 5, 3, 4 }, we consider length of LIS as three {1, 2, 5}, not as {1, 2, 3, 4} because we cannot make a strictly increasing array of integers with this LIS.

Given an array of n integers. Write a program to find minimum number of changes in array so that array is strictly increasing of integers. In strictly increasing array A[i] < A[i+1] for 0 <= i < n

```
// To find min elements to remove from array
// to make it strictly increasing
int minRemove(int arr[], int n)
{
    int LIS[n], len = 0;

    // Mark all elements of LIS as 1
    for (int i = 0; i < n; i++)
        LIS[i] = 1;

    // Find LIS of array
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j] && (i-j)<=(arr[i]-arr[j])){
                LIS[i] = max(LIS[i], LIS[j] + 1);
            }
        }
        len = max(len, LIS[i]);
    }

    // Return min changes for array
    // to strictly increasing
    return n - len;
}
```

```
class Solution{
    public:

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

1827. Minimum Operations to Make the Array Increasing

Easy 456 24 Add to List Share

You are given an integer array `nums` (**0-indexed**). In one operation, you can choose an element of the array and increment it by `1`.

- For example, if `nums = [1,2,3]`, you can choose to increment `nums[1]` to make `nums = [1,3,3]`.

Return *the **minimum** number of operations needed to make* `nums` ***strictly increasing***.

An array `nums` is **strictly increasing** if `nums[i] < nums[i+1]` for all `0 <= i < nums.length - 1`. An array of length `1` is trivially strictly increasing.

```
class Solution {
public:
    int minOperations(vector<int>& nums) {
        int res = 0, last = 0;
        for (auto n : nums) {
            res += max(0, last - n + 1);
            last = max(n, last + 1);
        }
        return res;
    }
};
```

Jump Game

Medium Accuracy: 49.57% Submissions: 20331 Points: 4

Given an positive integer N and a list of N integers A[]. Each element in the array denotes the maximum length of jump you can cover. Find out if you can make it to the last index if you start at the first index of the list.

Example 1:

```
Input:
N = 6
A[] = {1, 2, 0, 3, 0, 0}
Output:
1
Explanation:
Jump 1 step from first index to
second index. Then jump 2 steps to reach
4th index, and now jump 2 steps to reach
the end.
```

Example 2:

```
Input:
N = 3
A[] =  {1, 0, 2}
Output:
0
Explanation:
You can't reach the end of the array.
```

```cpp
class Solution {
  public:
    int canReach(int arr[], int n) {
        // code here
        int rechable =0;
        for(int i=0 ;i<n;i++)
        {
            if(rechable<i)
            {
                return false;
            }
            rechable=max(rechable,i+arr[i]);
        }
        return true;
    }
};
```

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, they cannot move through that element. If the end isn't reachable, return -1.

Method 1: Naive Recursive Approach.

Approach: A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.

minJumps(start, end) = Min (minJumps(k, end)) for all k reachable from start

Complexity Analysis:

- **Time complexity:** O(n^n).
  There are maximum n possible ways to move from a element. So maximum number of steps can be N^N so the upperbound of time complexity is O(n^n)
- **Auxiliary Space:** O(1).
  There is no space required (if recursive stack space is ignored).

Method 2: Dynamic Programming.

Approach:

1. In this way, make a jumps[] array from left to right such that jumps[i] indicate the minimum number of jumps needed to reach arr[i] from arr[0].
2. To fill the jumps array run a nested loop inner loop counter is j and outer loop count is i.
3. Outer loop from 1 to n-1 and inner loop from 0 to i.
4. if i is less than *j + arr[j]* then set jumps[i] to minimum of jumps[i] and jumps[j] + 1. initially set jump[i] to INT MAX
5. Finally, return jumps[n-1].

```cpp
int min(int x, int y) { return (x < y) ? x : y; }

// Returns minimum number of jumps
// to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    // jumps[n-1] will hold the result
    int* jumps = new int[n];
    int i, j;

    if (n == 0 || arr[0] == 0)
        return INT_MAX;

    jumps[0] = 0;

    // Find the minimum number of jumps to reach arr[i]
    // from arr[0], and assign this value to jumps[i]
    for (i = 1; i < n; i++) {
        jumps[i] = INT_MAX;
        for (j = 0; j < i; j++) {
            if (i <= j + arr[j] && jumps[j] != INT_MAX) {
                jumps[i] = min(jumps[i], jumps[j] + 1);
                break;
            }
        }
    }
    return jumps[n - 1];
}
```

DP APPROACH

Time Complexity: O(n^2)

Auxiliary Space: O(n)

```cpp
// Function to return the minimum number
// of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int n)
{
```

Recursive approach

```cpp
    // Base case: when source and
    // destination are same
    if (n == 1)
        return 0;

    // Traverse through all the points
    // reachable from arr[l]
    // Recursively, get the minimum number
    // of jumps needed to reach arr[h] from
    // these reachable points
    int res = INT_MAX;
    for (int i = n - 2; i >= 0; i--) {
        if (i + arr[i] >= n - 1) {
            int sub_res = minJumps(arr, i + 1);
            if (sub_res != INT_MAX)
                res = min(res, sub_res + 1);
        }
    }

    return res;
}
```

Method 3: Dynamic Programming.
In this method, we build jumps[] array from right to left such that jumps[i] indicates the minimum number of jumps needed to reach arr[n-1] from arr[i]. Finally, we return jumps[0].

There is a more smarter and optimal way of solving this in linear time Only. And on next page we will be discussing that

```cpp
int max(int x, int y)
{
    return (x > y) ? x : y;
}

// Returns minimum number of jumps
// to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{

    // The number of jumps needed to
    // reach the starting index is 0
    if (n <= 1)
        return 0;

    // Return -1 if not possible to jump
    if (arr[0] == 0)
        return -1;

    // initialization
    // stores all time the maximal
    // reachable index in the array.
    int maxReach = arr[0];

    // stores the number of steps
    // we can still take
    int step = arr[0];

    // stores the number of jumps
    // necessary to reach that maximal
    // reachable position.
    int jump = 1;

    // Start traversing array
    int i = 1;
    for (i = 1; i < n; i++) {
        // Check if we have reached the end of the array
        if (i == n - 1)
            return jump;

        // updating maxReach
        maxReach = max(maxReach, i + arr[i]);

        // we use a step to get to the current index
        step--;

        // If no further steps left
        if (step == 0) {
            // we must have used a jump
            jump++;

            // Check if the current index/position or lesser index
            // is the maximum reach point from the previous indexes
            if (i >= maxReach)
                return -1;

            // re-initialize the steps to the amount
            // of steps to reach maxReach from position i.
            step = maxReach - i;
        }
    }

    return -1;
}
```

O(n) time O(1) space

Approach

We have defined a max function Which will give max of two variables.

The main function will calculate the minimum number of jumps required to reach the end. This function will return an integer so its return type is INT. The parameters which we have passed in this function are the array and the size of the array n.

If n<=1 we will simply return 0 as no jumps are required as start=end.

If the first element of the array itself is 0 then return -1 this means you don't have enough power to jump to any cell.

Now, we define a variable maxReach which stores the maximum reachable index in the array of all times. Initially it is assigned to value of array indexed at 0. A[0]

The variable steps tells how many steps we can still take in order to reach the end. The jump variable stores the number of jumps necessary to reach that maximal reachable position in the array. Now we will start traversing the entire array from starting index I=1. Now we will check, whether we have reached the end of the array or not, or in simple words we will check whether I==n-1 in that case we will return jump, jump was preinitialised to 1 so we will return 1.

We will keep on updating the maxReach variable, it will be maximum of it's value, value at index I + index I

We will use a step variable to decrement the current index, as we have utilised step na. Now suppose step was 5 this means, we can take 1 step, 2 steps, 3 steps,,, at max 5 steps from the current index So by decrementing the step variable it means, we have utilised the step value. Now we check if our step variable becomes 0 this means we don't have any further steps left, this means, we now have to jump so we must have used a Jump, so we will increment jump. Do a jump++. And in the same if block, check if you have exceeded or reached MaxReach then in that case, we have to return -1.

Also now, reinitialise the step as maxReach-i because, these will be number of steps to reach maxReach from position i.

Return -1 in the end.

55. Jump Game

Medium 9533 555 Add to List Share

You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` *if you can reach the last index, or* `false` *otherwise.*

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

```cpp
class Solution {
public:
    bool canJump(vector<int>& A) {
        int i = 0;
        for (int reach = 0; i < A.size() and i <= reach; ++i)
            reach = max(i + A[i], reach);
        return i == A.size();
    }
};
```

909. Snakes and Ladders

Medium 459 95 Add to List Share

You are given an `n x n` integer matrix `board` where the cells are labeled from `1` to n^2 in a **Boustrophedon style** starting from the bottom left of the board (i.e. `board[n - 1][0]`) and alternating direction each row.

You start on square `1` of the board. In each move, starting from square `curr`, do the following:

- Choose a destination square `next` with a label in the range `[curr + 1, min(curr + 6, n^2)]`.
  - This choice simulates the result of a standard **6-sided die roll**: i.e., there are always at most 6 destinations, regardless of the size of the board.
- If `next` has a snake or ladder, you **must** move to the destination of that snake or ladder. Otherwise, you move to `next`.
- The game ends when you reach the square n^2.

A board square on row `r` and column `c` has a snake or ladder if `board[r][c] != -1`. The destination of that snake or ladder is `board[r][c]`. Squares `1` and n^2 do not have a snake or ladder.

Note that you only take a snake or ladder at most once per move. If the destination to a snake or ladder is the start of another snake or ladder, you do **not** follow the subsequent snake or ladder.

- For example, suppose the board is `[[-1,4],[-1,3]]`, and on the first move, your destination square is `2`. You follow the ladder to square `3`, but do **not** follow the subsequent ladder to `4`.

Return *the least number of moves required to reach the square* n^2. *If it is not possible to reach the square, return* `-1`.

Example 1:

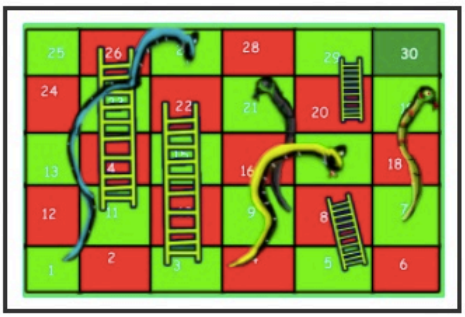

```
Input: board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]
Output: 4
Explanation:
In the beginning, you start at square 1 (at row 5, column 0).
You decide to move to square 2 and must take the ladder to square 15.
You then decide to move to square 17 and must take the snake to square 13.
You then decide to move to square 14 and must take the ladder to square 35.
You then decide to move to square 36, ending the game.
This is the lowest possible number of moves to reach the last square, so return 4.
```

Snake and Ladder Problem

Medium Accuracy: 49.35% Submissions: 11064 Points: 4

Given a **5x6** snakes and ladders board, find the minimum number of dice throws required to reach the destination or last cell (**30th** cell) from the source (1st cell).

You are given an integer **N** denoting the total number of snakes and ladders and an array **arr[]** of **2*N** size where **2*i** and **(2*i + 1)th** values denote the starting and ending point respectively of **ith** snake or ladder. The board looks like the following.



Example 1:

```
Input:
N = 8
arr[] = {3, 22, 5, 8, 11, 26, 20, 29,
       17, 4, 19, 7, 27, 1, 21, 9}
Output: 3
```

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

```
class Solution{
public:
    int minThrow(int N, int arr[]){
        int d[] = {+1, +2, +3, +4, +5, +6};
        vector<int> vis(31, 0);
        map<int,int> m;
        for(int i=0; i+1<2*N; i+=2) {
            m[arr[i]] = arr[i+1];
        }
        queue<pair<int,int>> q;
        vis[0] = vis[1] = 1;
        q.push({0, 1});
        while(!q.empty()) {
            int ans = q.front().first;
            int pos = q.front().second;
            q.pop();
            if(pos==30) return ans;
            for(int i=0; i<6; i++) {
                int newPos = pos + d[i];
                if(newPos>=0 && newPos<=30 && vis[newPos]==0) {
                    if(m.find(newPos)!=m.end() && m[newPos]>=0 && m[newPos]<=30 && vis[m[newPos]]==0) {
                        vis[m[newPos]] = 1;
                        q.push({ans+1, m[newPos]});
                    }
                    vis[newPos] = 1;
                    q.push({ans+1, newPos});
                }
            }
        }

        return -1;
    }
};
```

For every cell

As n=30 we create a visited vector of n+1 preinitialised with 0

A map of int,int

As the size of the array is 2N, 2i means start, 2i+1 means end index of snake I/ladder I

So for every snake/ladder, we are storing the ending index of it in the map at key start index

We have defined a pair of int and int and the queue elements will be go this type only

Initiallv both the cells are visited.

Push the 0 as the number of steps and 1 as the position in the queue first element

Now, until the queue is empty, do something

Store the respective values of the first queue element in ans,pos

Remove the first element from the queue as we no longer require it.

If you have reached the last cell, return the answer and this is the breakpoint of while loop

For each cell in a particular row, find where you will go and store it in newPos

If it is between bounds and has not been visited then

If the element is found in the map and its end is between bounds and it's end has not been visited, then mark the end as visited and then push 1 more roll dice and its new position where it will land in the queue

If it has has been visited at end, this means it is ladder, so for snake at this pos Mark the start point as visited and push the ans+1 and new Pos

```
class Solution {
public:
    int snakesAndLadders(vector<vector<int>>& board) {
        int n=board.size(); //this is the size of the board
        vector<vector<bool>> visited(n , vector<bool>(n,false)); //this is the 2d array for visited.
        queue<int> q; //this is the queue for bfs traversal.
        q.push(1); //we have to push 1 onto the queue.
        visited[n-1][0]=true; //set the first entry of the last row as true.
        int steps=0; //initialise the steps as 0 initially.
        while(!q.empty()){ //till the queue is not empty
            int size=q.size(); //this is the size of the queue.
            while(size--){  /// checking levels for smallest steps , similar to shortest path
                int currpos = q.front(); //this is the front of the queue.
                if(currpos==n*n) return steps; //if we have reached the end of the board, then we should return the steps.
                q.pop();  //we will pop the queue, as we no longer need it.
                for(int i=1;i<=6;i++){    /// a dice can get 1 to number 6
                    int nextpos=currpos+i; //this will be the current next position.
                    if(nextpos>n*n) break; //if the next pos has crossed
                    int r = n - (nextpos-1)/n -1;   // getting row of board matrix
                    int c = (nextpos-1)%n;           // getting column of board matrix
                    if(r%2 == n%2)      // this step is imp because the value after n will come just above n like 7 will
                        c = n-c-1; // come just above 6 and not above 1  , if we are given in other format we can skip this
                    if(!visited[r][c]){ //if not visited
                        visited[r][c]=true; //mark it as visited.
                        if(board[r][c]!=-1) q.push(board[r][c]);  // if it is ladder or snake push that value else push next pos
                        else q.push(nextpos);
                    }
                }
            }
            steps++;
        }
        return -1;
    }
};
```

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.
If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search of the graph.
Following is the implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0... N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

Matrix Chain Multiplication
Hard Accuracy: 59.72% Submissions: 23586 Points: 8

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrices are given in an array **arr[]** of size **N** (such that N = number of matrices + 1) where the **i^th** matrix has the dimensions **(arr[i-1] x arr[i])**.

Example 1:

```
Input: N = 5
arr = {40, 20, 30, 10, 30}
Output: 26000
Explaination: There are 4 matrices of dimension
40x20, 20x30, 30x10, 10x30. Say the matrices are
named as A, B, C, D. Out of all possible combinations,
the most efficient way is (A*(B*C))*D.
The number of operations are -
20*30*10 + 40*20*10 + 40*10*30 = 26000.
```

Example 2:

```
Input: N = 4
arr = {10, 30, 5, 60}
Output: 4500
Explaination: The matrices have dimensions
10*30, 30*5, 5*60. Say the matrices are A, B
and C. Out of all possible combinations,the
most efficient way is (A*B)*C. The
number of multiplications are -
10*30*5 + 10*5*60 = 4500.
```

```
class Solution
{
public:
    int  dp[1001][1001];
    int solve(int a[], int i, int j)
    {
        if (i==j) return 0;
        if (dp[i][j]!=-1) return dp[i][j];
        dp[i][j] = INT_MAX;
        for (int k = i; k < j; k++)
        {
            dp[i][j] = min(dp[i][j] , solve(a,i,k)  +
                                      solve(a,k+1,j)  +
                                      a[i-1]*a[k]*a[j]);
        }
        return dp[i][j];
    }
    int matrixMultiplication(int n, int a[])
    {
         memset(dp, -1, sizeof dp);
        return solve(a,1,n-1);
    }
};
```

In this matrix chain multiplication we can use simple recursion in order to solve the main problem as this is the Stadard dynamic problem problem where we have optimal substructure and overlapping subproblem.
The top down approach is pretty straight forward, we create a 2D DP memo.
The parameters of the recursive function will be the Array which we are passing, the start index and the end index So if both the indices are pointing to the same place, then return 0. If the Dp has already been updated, and is no Longer -1, then in that case, we have to return the new value itself. Initialise the Dp to INTMAX
now traverse all the rows and t hen simply update the Dp as min of the sum of solve of i,k and solve on k+1,j and the third entry would be multiple of previous array element times a[k] times a[j].

Memoization

Just adding a few lines to the above code, we can reduce time complexity from exponential to cubi

```
int dp[501][501];

    int solve(int i, int j,int arr[])
    {
        if(i>=j)return 0;

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

        int ans=INT_MAX;
        for(int k=i;k<=j-1;k++)
        {
            int tempAns = solve(i,k,arr) + solve(k+1,j,arr)
                          + arr[i-1]*arr[k]*arr[j];

            ans=min(ans,tempAns);

        }
        return dp[i][j] = ans;
    }
    int matrixMultiplication(int N, int arr[])
    {
        memset(dp,-1,sizeof(dp));
        return solve(1,N-1,arr);
    }
    ```
```

int MatrixChainOrder(int arr[], int n)

{

```
/* For simplicity of the program, one
extra row and one extra column are
allocated in arr[][]. 0th row and 0th
column of arr[][] are not used */
int dp[n][n];

int i, j, k, L, q;

/* dp[i, j] = Minimum number of scalar
multiplications needed to compute the
matrix A[i]A[i+1]...A[j] = A[i..j] where
dimension of A[i] is arr[i-1] x arr[i] */

// cost is zero when multiplying
// one matrix.
for (i = 1; i < n; i++)
    dp[i][i] = 0;

// L is chain length.
for (L = 2; L < n; L++)
{
    for (i = 1; i < n - L + 1; i++)
    {
        j = i + L - 1;
        dp[i][j] = INT_MAX;
        for (k = i; k <= j - 1; k++)
        {
            // q = cost/scalar multiplications
            q = dp[i][k] + dp[k + 1][j]
                + arr[i - 1] * arr[k] * arr[j];
            if (q < dp[i][j])
                dp[i][j] = q;
        }
    }
}

return dp[1][n - 1];
```

}
```

If you have seen Balloon Burst and this problem, not able to find the solution .
Just read the following pattern carefully .

These both are the child problem of MCM .

Problem : If a chain of matrices is given, we have to find the minimum number of the correct sequence of matrices to multiply.

The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.
Note:no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...
However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
Clearly the first parenthesization requires less number of operations.
Note ; We'll be given an array arr[ ] which represents the chain of matrices such that the ith matrix arr[i] is of dimension arr[i-1] x arr[i].
That's why we start out 'k' i.e partition from 'i' =1 so that arr[1] is of dimentions arr[1-1] * arr[1] else we'll get index out of bound error Eg arr[0-1] * arr[0] is not possible
So first half of the array is from i to k & other half is from k+1 to j
Also we need to find the cost of multiplication of these 2 resultant matrixes (first half & second half) which is nothing but arr[i-1] * arr[k] * arr[j]

Recursion
Here is the recursive algo :

Form a palindrome

Medium Accuracy: 53.0% Submissions: 13460 Points: 4

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.
For Example:
ab: Number of insertions required is 1. **b**ab or aba
aa: Number of insertions required is 0. aa
abcd: Number of insertions required is 3. **dcb**abcd

Example 1:

```
Input: str = "abcd"
Output: 3
Explanation: Inserted character marked
with bold characters in dcbabcd
```

Example 2:

```
Input: str = "aa"
Output: 0
Explanation:"aa" is already a palindrome.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **countMin()** which takes the string **str** as inputs and returns the answer.

Expected Time Complexity: $O(N^2)$, N = |str|
Expected Auxiliary Space: $O(N^2)$

Constraints:
$1 \le |str| \le 10^3$
str contains only lower case alphabets.

View Bookmarked Problems

Company Tags

```cpp
// } Driver Code Ends
//User function template for C++

class Solution{
  public:
    int LCS(string A,string B)
  {
      int n = A.length();
      int m = B.length();
      int maxx=0;

      int dp[n+1][m+1];
      for(int i=0;i<=n;i++)
      {
          for(int j=0;j<=m;j++)
          {
              if(i==0 || j==0)
              {
                  dp[i][j] = 0;
              }
              else if(A[i-1]==B[j-1])
              {
                  dp[i][j] = 1+dp[i-1][j-1];
                  maxx = max(maxx,dp[i][j]);
              }
              else
              {
                  dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
              }
          }
      }
      return maxx;
  }
    int countMin(string str){

    string B = str;
    reverse(B.begin(),B.end());

    int num = LCS(str,B);
    return str.length()-num;
    }
};
```

32. Longest Valid Parentheses

Hard 6922 240 Add to List Share

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

Example 1:

```
Input: s = "(()"
Output: 2
Explanation: The longest valid parentheses substring is "
()".
```

Example 2:

```cpp
class Solution
{
    public:
    int longestValidParentheses(string s)
    {
        if (s.size() == 0) return 0;
        int res = 0;
        vector<int> dp (s.size(), 0);
        for (int i=1; i<s.size(); ++i)
        {
            if(s[i]==')')
                if(s[i-1] == '(' )
                    dp [i] = (i >=2 ? dp[i-2] : 0) + 2;
                else if(i-dp[i-1]>0&&s[i-dp[i-1]-1]=='(')
                    dp[i]=dp[i-1]+2+((i-dp[i-1])>=2?dp[i-dp[i-1]-2]:0);
            res = dp[i] > res ? dp[i] : res;
        }
        return res;
    }
};
```

Longest valid Parentheses

Hard Accuracy: 48.35% Submissions: 14592 Points: 8

Given a string **S** consisting of opening and closing parenthesis '**(**' and '**)**'. Find length of the longest valid parenthesis substring.

A parenthesis string is valid if:

- For every opening parenthesis, there is a closing parenthesis.
- Opening parenthesis must be closed in the correct order.

Example 1:

```
Input: S = (((
Output: 2
Explaination: The longest valid
parenthesis substring is "()".
```

Example 2:

```
Input: S = )()())
Output: 4
Explaination: The longest valid
parenthesis substring is "()()".
```

Your Task:
You do not need to read input or print anything. Your task is to complete the function **maxLength()** which takes string S as input parameter and returns the length of the maximum valid parenthesis substring.

Expected Time Complexity: O(|S|)
Expected Auxiliary Space: O(|S|)

Constraints:
$1 \le |S| \le 10^5$

```cpp
// } Driver Code Ends
// User function Template for C++

class Solution
{
    public:
    int maxLength(string s)
    {
        if (s.size() == 0) return 0;
        int res = 0;
        vector<int> dp (s.size(), 0);
        for (int i=1; i<s.size(); ++i)
        {
            if(s[i]==')')
                if(s[i-1] == '(' )
                    dp [i] = (i >=2 ? dp[i-2] : 0) + 2;
                else if(i-dp[i-1]>0&&s[i-dp[i-1]-1]=='(')
                    dp[i]=dp[i-1]+2+((i-dp[i-1])>=2?dp[i-dp[i-1]-2]:0);
            res = dp[i] > res ? dp[i] : res;
        }
        return res;
    }
};

} // } Driver Code Ends
```

Explanation of the DP solution. When the question comes of parentheses, many a times, stack solution seems much more easier. For this question also, stack solution is very easy and intuitive to understand, but if you see, there exists an optimal substructure for this problem, therefore we are using DP to solve this problem and there are overlapping subproblems, whose result needs to be stored in a memo.

```
class Solution
{
   public:
   int maxLength(string s)
   {
//base case
//check if the string is empty, return 0
        if (s.size() == 0) return 0;
// pre-initialise the res variable with 0 which will store the result which we will return at the end.
       int res = 0;
//declare an empty DP vector whose size is same as the length of the string and preinitialise it with 0.
       vector<int> dp (s.size(), 0);
//start the for loop from i=1 to length of string-1 and then do something
       for (int i=1; i<s.size(); ++i)
       {
//check if current element is the closing index
           if(s[i]==')')
//check if the previous index was an opening index then
               if(s[i-1] == '(' )
//if your I is greater than 2 then dp[I] = dp[i-2]+2 otherwise dp[I] = 2
/*
This means that if you are more then the index 2, then its answer will be stored in dp[i-2] and now you will add 2, because
you have found another valid answer and that valid answer is ( at index i-1 and ) at index I. so at index I you will store, whatever is presen
*/
                   dp [i] = (i >=2 ? dp[i-2] : 0) + 2;
/*
The else if tells that current s[I] is ) then check that if index i minus dp[i-1] is greater than 0 this means suppose current index is 3 and
other condition is if s[i-dp[i-1]] is an opening bracket this means, currently at the index i-dp[i-1] you have an opening bracket then you do
*/
               else if(i-dp[i-1]>0&&s[i-dp[i-1]-1]=='(')
// if that index i-dp[i-1] is greater than equal to 2 then you simply assign dp[I] as dp[i-dp[i-1]-2]  otherwise you mark it as 0.
                   dp[i]=dp[i-1]+2+((i-dp[i-1])>=2?dp[i-dp[i-1]-2]:0);
//at the end you update the res, based on max(dp[I],res)
           res = dp[i] > res ? dp[i] : res;
      }
      return res;
  }
```

Find all possible palindromic partitions of a String

Hard Accuracy: 65.73% Submissions: 6901 Points: 8

Given a String **S**, Find all possible Palindromic partitions of the given String.

Example 1:

Google, Facebook, Microsoft

Backtracking

Dynamic Programming

```
Input:
S = "geeks"
Output:
g e e k s
g ee k s
Explanation:
All possible palindromic partitions
are printed.
```

Example 2:

```
Input:
S = "madam"
Output:
m a d a m
m ada m
madam
```

gfg hard

Your Task:
You don't need to read input or print anything. Your task is to complete the function **allPalindromicPerms()** which takes a String S as input parameter and returns a list of lists denoting all the possible palindromic partitions in the order of their appearance in the original string.

Expected Time Complexity: $O(N*2^N)$
Expected Auxiliary Space: $O(N^2)$, where N is the length of the String

```cpp
class Solution {
  public:
  bool isPalindrome(string s){
      int i = 0, j = s.size()-1;
      while(i<=j){
          if(s[i]==s[j]){
              i++, j--;
              continue;
          }
          return false;
      }
      return true;
  }
  void helper(string &S, vector<vector<string>>&res, int i,vector<string>&copy){
      if(i>=S.length()){
          res.push_back(copy);
          return;
      }
      string str = "";
      for(int j = i; j<S.size(); j++){
          str += S[j];
          if(isPalindrome(str)){
              copy.push_back(str);
              helper(S,res,j+1,copy);
              copy.pop_back();
          }
      }

  }
    vector<vector<string>> allPalindromicPerms(string S) {
        // code here
        vector<vector<string>>res;
        //string temp = "";
        vector<string> copy;
        helper(S,res,0,copy);
        return res;
    }
};
```

bool isPal(String s, int l, int h)
↳ checks if string starting from index l & ending at index h is palindrome or not
↳ return type

Input: nitin
output: n i t i n
n iti n
nitin

length

```
f( vvs &Result , vs&Part , int start, int n, String S ){
    if (start >= n) { Result.push_back(Part); return; }
    for( i=start; i<n; i++ )
    {
        if isPal( S, start, i )
        {
            Part.push_back(S.substr(start, i-start+1))
            f( Result , Part, i+1, n, S )
            Part.pop_back()
```

```
int32_main()
{
    f(Result, Part, 0, n, S);
    for(i=0; i<Result.size(); i++){
        for(j=0; j< Result[i].size(); j++)
            cout<< Result[i][j] << " ";
    cout<<endl;
}
```

Subset Sum Problem

Medium Accuracy: 51.38% Submissions: 41623 Points: 4

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

Example 1:

```
Input:
N = 6
arr[] = {3, 34, 4, 12, 5, 2}
sum = 9
Output: 1
Explanation: Here there exists a subset with
sum = 9, 4+3+2 = 9.
```

Example 2:

```
Input:
N = 6
arr[] = {3, 34, 4, 12, 5, 2}
sum = 30
Output: 0
Explanation: There is no subset with sum 30.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **isSubsetSum()** which takes the array arr[], its size N and an integer **sum** as input parameters and returns boolean value true if there exists a subset with given sum and false otherwise.
The driver code itself prints 1, if returned value is true and prints 0 if returned value is false.

Expected Time Complexity: O(sum*N)
Expected Auxiliary Space: O(sum*N)

```cpp
//This question is similar to knapsack problem
//where sum= total weigth and weight arr = arr of element

class Solution{
public:
    bool isSubsetSum(int n, int arr[], int sum)
    {
        bool dp[n+1][sum+1];
        for(int i=0;i<=n;i++)
        {
            for(int j=0;j<=sum;j++)
            {
                if(!i)
                    dp[i][j] = false;
                if(!j)
                    dp[i][j] = true;
            }
        }
        for(int i=1;i<=n;i++)
            for(int j=1;j<=sum;j++)
                if(arr[i-1]<=j) dp[i][j] = dp[i-1][j-arr[i-1]]||dp[i-1][j];
                else dp[i][j] = dp[i-1][j];
        return dp[n][sum];
    }
};
```

Brackets in Matrix Chain Multiplication

Medium Accuracy: 70.96% Submissions: 1147 Points: 4

Given an array **p[]** of length **n** used to denote the dimensions of a series of matrices such that dimension of **i'th** matrix is **p[i] * p[i+1]**. There are a total of **n-1** matrices. Find the most efficient way to multiply these matrices together.
The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications such that you need to perform minimum number of multiplications. There are many options to multiply a chain of matrices because matrix multiplication is associative i.e. no matter how one parenthesize the product, the result will be the same.

Example 1:

```
Input:
n = 5
p[] = {1, 2, 3, 4, 5}
Output: (((AB)C)D)
Explaination: The total number of
multiplications are (1*2*3) + (1*3*4)
+ (1*4*5) = 6 + 12 + 20 = 38.
```

Example 2:

```
Input:
n = 3
p = {3, 3, 3}
Output: (AB)
Explaination: The total number of
multiplications are (3*3*3) = 27.
```

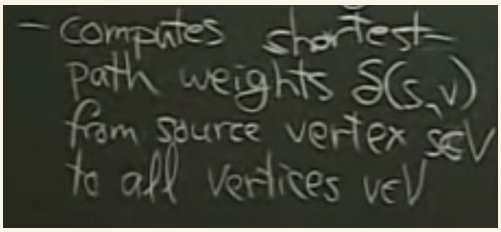
Expected Time Complexity: $O(n^3)$
Expected Auxiliary Space: $O(n^2)$

```cpp
class Solution {

    // Cache
    map<string, pair<int, string>> dp;

    public:
    string matrixChainOrder(int A[], int n)
    {
        return f(A, 0, n - 1).second;
    }
    pair<int, string> f(int p[], int l, int r)
    {
        // Base case
        /*
        For [1, 2, 3], base case will hit when l = 0, r = 1 or l = 1, r = 2
        or we can say we are at matrix A or B
        l = 0 -> A
        l = 1 -> B
        so on..
        */
        if(l + 1 == r) return {0, "" + string(1, l + 'A')};
        string key = to_string(l) + ";" + to_string(r);
        if(dp.count(key)) return dp[key];

        int currMin = INT_MAX;
        string minString;
        for(int k = l + 1; k < r; k++) {
            auto p1 = f(p, l, k), p2 = f(p, k, r);
            if(p1.first + p2.first + p[l] * p[k] * p[r] < currMin)
            {
                currMin = p1.first + p2.first + p[l] * p[k] * p[r];
                // Multiply the 2 strings
                minString = p1.second + p2.second;
            }
        }
        // Enclose them in brackets and return
        return dp[key] = {currMin, "(" + minString + ")"};
    }
};
```

Bellman Ford Problem

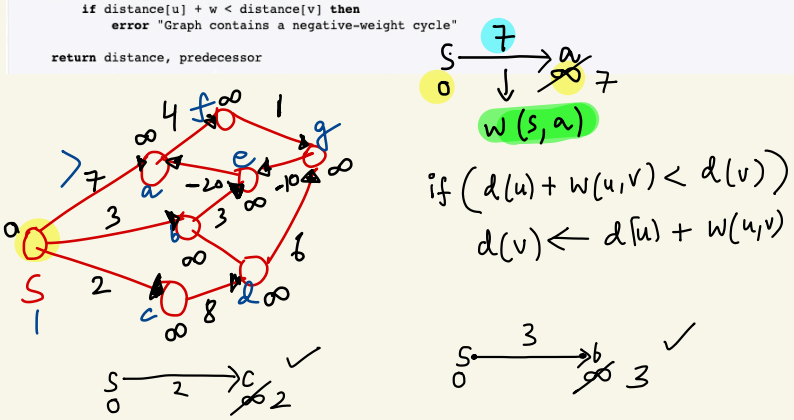
Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O((V+E)LogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

Reports the negative weight cycle if it exists



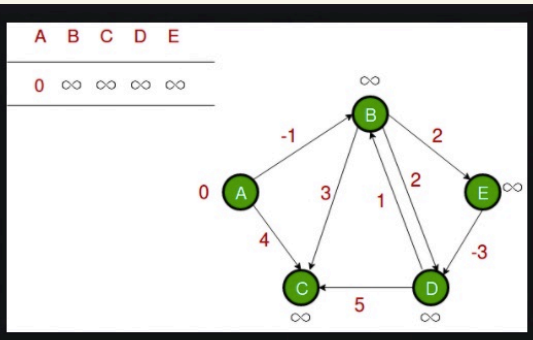

```
function BellmanFord(list vertices, list edges, vertex source) is

    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance := list of size n
    predecessor := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do

        distance[v] := inf             // Initialize the distance to all vertices to infinity
        predecessor[v] := null         // And having a null predecessor

    distance[source] := 0              // The distance from the source to itself is, of course, zero

    // Step 2: relax edges repeatedly

    repeat |V|-1 times:
         for each edge (u, v) with weight w in edges do
             if distance[u] + w < distance[v] then
                 distance[v] := distance[u] + w
                 predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"

    return distance, predecessor
```

Algorithm
Following are the detailed steps.
Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.
1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
.....**a)** Do following for each edge u-v
.....If dist[v] > dist[u] + weight of edge uv, then update dist[v]
.....dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
.....If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

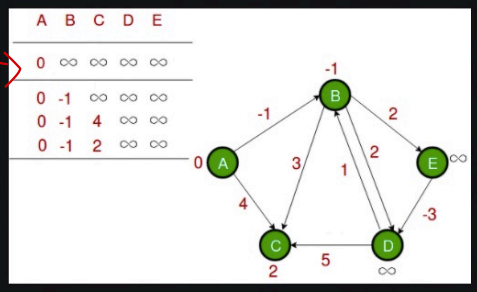


How does this work? Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| - 1 edges in any simple path, that is why the outer loop runs |v| - 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer this or MIT Video Lecture)

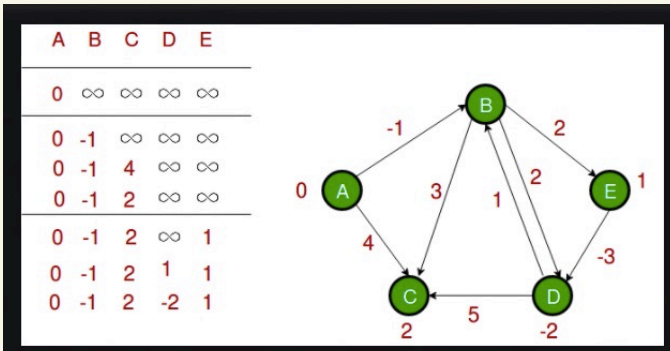
Example
Let us understand the algorithm with following example graph. The images are taken from this source.
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Notes
1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
2) Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijkstra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.
3) Bellman-Ford does not work with undirected graph with negative edges as it will declared as negative cycle.

Exercise
1) The standard Bellman-Ford algorithm reports the shortest path only if there are no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
2) Can we use Dijkstra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijkstra's algorithm for the modified graph. Will this algorithm work?

Bellman-Ford Analysis

```
for v in V:                                   } O(V)
    v.d = ∞
    v.π = None
s.d = 0
for i from 1 to |V| − 1:                      } O(VE)
    for (u, v) in E:                          } O(E)
        relax(u, v)                           } O(1)
for (u, v) in E:                              } O(E)
    if v.d > u.d + w(u, v):
        report that a negative-weight cycle exists
```

TOTAL: O(VE)

```cpp
#define vi vector<int>
#define vvi vector<vi>
class Solution
{
    public:
    int isNegativeWeightCycle(int n, vvi edges)
    {
        vi d(n, INT_MAX);
        d[0] = 0;
        for(int i=1; i<=n-1; i++)
            for(auto& e: edges)
            {
                if(d[e[0]]==INT_MAX) continue;
                if(d[e[0]] + e[2] < d[e[1]]) d[e[1]] = d[e[0]] + e[2];
            }

        for(auto& e: edges) if(d[e[0]] + e[2] < d[e[1]]) return 1;

        return 0;
    }
};
```

Negative weight cycle

Medium Accuracy: 50.77% Submissions: 16587 Points: 4

Given a weighted directed graph with n nodes and m edges. Nodes are labeled from 0 to n-1, the task is to check if it contains a negative weight cycle or not.
Note: edges[i] is defined as u, v and weight.

Example 1:

```
Input: n = 3, edges = {{0,1,-1},{1,2,-2},
{2,0,-3}}
Output: 1
Explanation: The graph contains negative weight
cycle as 0->1->2->0 with weight -1,-2,-3,-1.
```

Example 2:

```
Input: n = 3, edges = {{0,1,-1},{1,2,-2},
{2,0,3}}
Output: 0
Explanation: The graph does not contain any
negative weight cycle.
```

Your Task:
You don't need to read or print anyhting. Your task is to complete the function **isNegativeWeightCycle()** which takes n and edges as input paramater and returns 1 if graph contains negative weight cycle otherwise returns 0.

Expected Time Complexity: O(n*m)
Expected Space Compelxity: O(n)

Word Wrap Problem

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.
The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

```cpp
#include<bits/stdc++.h>
#define vi vector<int>
#define lli long long int
#define inf INT_MAX
lli dp[501];
class Solution
{
    public:
    lli f(int index, int k, vi &nums, int pre)
    {
        int n = nums.size();
        if(index==n) return (-1 * pre);
        if(dp[index]!=-1) return dp[index];
        int prev = 0;
        lli ans = inf;
        for(int i = index; i < n; i++)
        {
            prev += nums[i];
            if(prev > k) break;
            int x = pow(k-prev,2);
            lli temp = x + f(i+1, k, nums, x);
            ans = min(ans, temp);
            prev += 1;
        }
        return dp[index]=ans;
    }

    int solveWordWrap(vi &nums, int k)
    {
        memset(dp, -1, sizeof(dp));
        return f(0, k,nums,0);
    }
};
```

Word Wrap

Medium Accuracy: 47.35% Submissions: 11583 Points: 4

Given an array **nums[]** of size **n**, where **nums[i]** denotes the number of characters in one word. Let **K** be the limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width. When line breaks are inserted there is a possibility that extra spaces are present in each line. The extra spaces include spaces put at the end of every line **except the last one**.

You have to **minimize** the following total cost where **total cost** = Sum of cost of all lines, where cost of line is = (Number of extra spaces in the line)^2.

Example 1:

Input: nums = {3,2,2,5}, k = 6
Output: 10
Explanation: Given a line can have 6 characters,
Line number 1: From word no. 1 to 1
Line number 2: From word no. 2 to 3
Line number 3: From word no. 4 to 4
So total cost = $(6-3)^2 + (6-2-2-1)^2 = 3^2+1^2 = 10$.
As in the first line word length = 3 thus extra spaces = 6 - 3 = 3 and in the second line there are two word of length 2 and there already 1 space between two word thus extra spaces = 6 - 2 -2 -1 = 1. As mentioned in the problem description there will be no extra spaces in the last line. Placing first and second word in first line and third word on second line would take a cost of $0^2 + 4^2 = 16$ (zero spaces on first line and 6-2 = 4 spaces on second), which isn't the minimum possible cost.

We will solve this question using the top down memorisation approach where we will use the helper function f which will recursively call itself, and before returning anything we will store the answer in the dp table, and if the answer is already exist in the dp table, we won't go and recurse, instead we will simply return what is stored in the dp table.

So we will have n<=500 so we will create a dp array of size 500+1 and preinitialise it with -1. And then we will return f(0,k,nums,0)
The helper function f has 4 parameters, first parameter is the starting index I and the second parameter is k which is given in the question that we have size k for each line. A sentence can be formed at max of k words in a particular line. The third parameter is the nums array and the fourth parameter is the pre which is the error rate initially it is 0.

Let us look into the helper function. The helper function first computes n as size of the array and then we check if the index has reached the size of the array in that case we will return -1*pre. That is the base case of the recursion. Now we will check if the value of dp is not equal to -1 at index then in that case, it means, the result has changed at the index so instead of recomputing, we return whatever is stored at dp[index]. Now we have a variable prev and we initialise it to 0. We have an answer whose type is long long int. It is preinitialised to infinity. Now we run a for loop that runs from index to the size of the array. Each time, we increase the prev by storing the value at index I of nums. Suppose nums[I]=3 then prev+=nums[I] means we have added the value to it. Immediately after adding . Check if the value of prev exceeds k. If that is the case, we have to break as there is not a space for another word to be stored in the line. Now, x is the error rate at each cell which is k-prev raised to the power 2.
We will store the value temp which will be x+ whatever the helper function returns on index I+1, k, nums,x
After all the recursive calls, answer will be minimum of ans and temp. And we will increment prev as we have to move diagonally.

That is the power of recursion. At the end return dp[index] = answer. As we will be needing this answer again and again.

Painter's Partition Problem

Expected Time Complexity: O(n log m) , m = sum of all boards' length
Expected Auxiliary Space: O(1)

ASKED IN CODENATION, GOOGLE, MICROSOFT

The Painter's Partition Problem-II

Hard Accuracy: 48.95% Submissions: 21309 Points: 8

Dilpreet wants to paint his dog's home that has **n** boards with different lengths. The length of i[th] board is given by **arr[i]** where **arr[]** is an array of **n** integers. He hired **k** painters for this work and each painter takes **1 unit time to paint 1 unit of the board.**

The problem is to find the minimum time to get this job done if all painters start together with the constraint that any painter will only paint continuous boards, say boards numbered **{2,3,4}** or only board **{1}** or nothing but not boards **{2,4,5}**.

Example 1:

```
Input:
n = 5
k = 3
arr[] = {5,10,30,20,15}
Output: 35
Explanation: The most optimal way will be:
Painter 1 allocation : {5,10}
Painter 2 allocation : {30}
Painter 3 allocation : {20,15}
Job will be done when all painters finish
i.e. at time = max(5+10, 30, 20+15) = 35
```

Example 2:

```
Input:
n = 4
k = 2
arr[] = {10,20,30,40}
Output: 60
Explanation: The most optimal way to paint:
Painter 1 allocation : {10,20,30}
Painter 2 allocation : {40}
Job will be complete at time = 60
```

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}.

We have n boards of different length. arr[i] stores the length of ith board.
We have k painters and each painter takes 1 unit of time to paint 1 unit of the board.
What will be the minimum time to finish painting all the blocks

From the above examples, it is obvious that the strategy of dividing the boards into k equal partitions won't work for all the cases. We can observe that the problem can be **broken down** into: Given an array A of non-negative integers and a positive integer k, we have to divide A into k of fewer partitions such that the maximum sum of the elements in a partition, overall partitions is minimized. So for the second example above, possible **divisions** are:

* One partition: so time is 100.

* Two partitions: (10) & (20, 30, 40), so time is 90. Similarly we can put the first divider after 20 (=> time 70) or 30 (=> time 60); so this means the minimum time: (100, 90, 70, 60) is 60.

A **brute force** solution is to consider all possible set of contiguous partitions and calculate the maximum sum partition in each case and return the minimum of all these cases.

1) Optimal Substructure:
We can implement the naive solution using recursion with the following optimal substructure property:
Assuming that we already have k-1 partitions in place (using k-2 dividers), we now have to put the k-1 th divider to get k partitions.
How can we do this? We can put the k-1 th divider between the i th and i+1 th element where i = 1 to n. Please note that putting it before the first element is the same as putting it after the last element.
The total cost of this arrangement can be calculated as the **maximum** of the following:
a) The cost of the last partition: sum(Ai..An), where the k-1 th divider is before element i.
b) The maximum cost of any partition already formed to the left of the k-1 th divider.
Here a) can be found out using a simple **helper function** to calculate sum of elements between two indices in the array. How to find out b) ?
We can observe that b) actually is to place the k-2 separators as fairly as possible, so it is a **subproblem** of the given problem. Thus we can write the optimal

$$T(n,k) = min\left\{ max_{i=1}^{n}\left\{ T(i,k-1), \sum_{j=i+1}^{n} A_j \right\}\right\}$$

The base case are:

$$T(1,k) = A1$$

$$T(n,1) = \sum_{i=1}^{n} A_i$$

```
// function to calculate sum between two indices
// in array
int sum(int arr[], int from, int to)
{
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

// for n boards and k partitions
int partition(int arr[], int n, int k)
{
    // base cases
    if (k == 1) // one partition
        return sum(arr, 0, n - 1);
    if (n == 1)  // one board
        return arr[0];

    int best = INT_MAX;

    // find minimum of all possible maximum
    // k-1 partitions to the left of arr[i],
    // with i elements, put k-1 th divider
    // between arr[i-1] & arr[i] to get k-th
    // partition
    for (int i = 1; i <= n; i++)
        best = min(best, max(partition(arr, i, k - 1),
                             sum(arr, i, n - 1)));

    return best;
}
```

This Recursive solution is exponential in nature.

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

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

    // n=1
    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];

    // 2 to k partitions
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {

            // track minimum
            int best = INT_MAX;

            // i-1 th separator before position arr[p=1..j]
            for (int p = 1; p <= j; p++)
                best = min(best, max(dp[i - 1][p],
                                     sum(arr, p, j - 1)));

            dp[i][j] = best;
        }
    }

    // required
    return dp[k][n];
}
```

```
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];
```

K n ^ 3

Bottum up DP

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:

C++ Java C#

```
int sum[n+1] = {0};

// sum from 1 to i elements of arr
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]));
```

k n ^2

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:

C++ Java C# Javascript

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

Binary search approach — — — — — — — — — — — — — — O(nlogm)>

```
int getMax(int arr[], int n)
{
    int max = INT_MIN;
    for (int i = 0; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// return the sum of the elements in the array
int getSum(int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    return total;
}

// find minimum required painters for given maxlen
// which is the maximum length a painter can paint
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) {

            // for next count
            total = arr[i];
            numPainters++;
        }
    }

    return numPainters;
}
```

Allocate minimum number of pages

Hard Accuracy: 48.87% Submissions: 37661 Points: 8

You are given **N** number of books. Every **ith** book has **Ai** number of pages and are arranged in **sorted order**.

You have to allocate contagious books to **M** number of students. There can be many ways or permutations to do so. In each permutation, one of the M students will be allocated the maximum number of pages. Out of all these permutations, the task is to find that particular permutation in which the maximum number of pages allocated to a student is minimum of those in all the other permutations and print this minimum value.

Each book will be allocated to exactly one student. Each student has to be allocated at least one book.

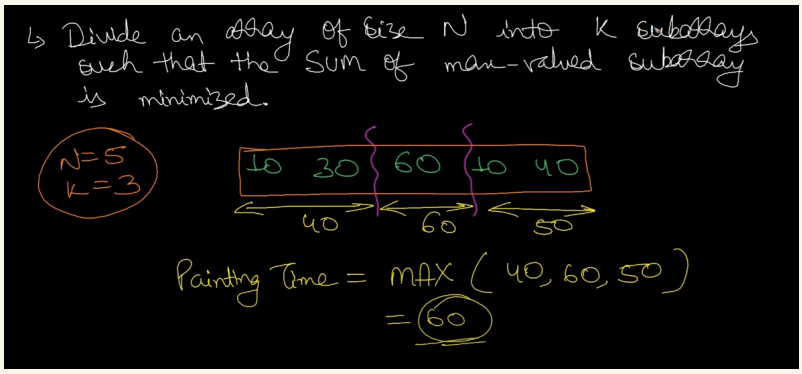
Note: Return **-1** if a valid assignment is not possible, and allotment should be in contiguous order (see the explanation for better understanding).

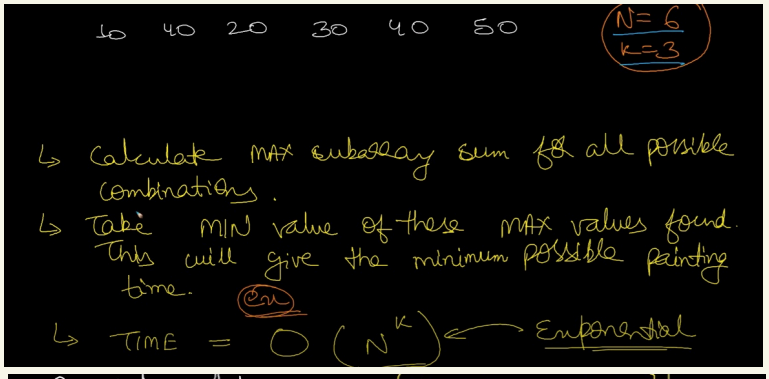
Example 1:

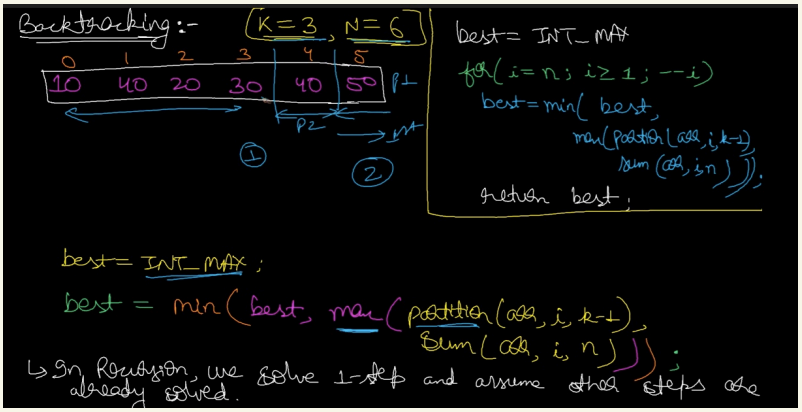
```
Input:
N = 4
A[] = {12,34,67,90}
M = 2
Output:113
Explanation:Allocation can be done in
following ways:{12} and {34, 67, 90}
Maximum Pages = 191{12, 34} and {67, 90}
Maximum Pages = 157{12, 34, 67} and {90}
Maximum Pages =113. Therefore, the minimum
of these cases is 113, which is selected
as the output.
```

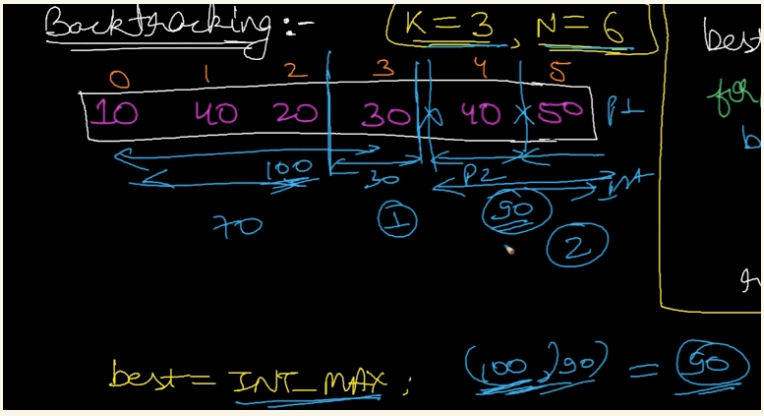
```
int partition(int arr[], int n, int k)
{
    int lo = getMax(arr, n);
    int hi = getSum(arr, n);

    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        int requiredPainters = numberOfPainters(arr, n, mid);

        // find better optimum in lower half
        // here mid is included because we
        // may not get anything better
        if (requiredPainters <= k)
            hi = mid;

        // find better optimum in upper half
        // here mid is excluded because it gives
        // required Painters > k, which is invalid
        else
            lo = mid + 1;
    }

    // required
    return lo;
}
```

```
class Solution
{
   public:
   int findPages(int A[], int N, int M)
   {
       int res=-1;
       int sum=0;
       int start=A[0];
       for(int i=0;i<N;++i){
           sum+=A[i];
           start=max(start,A[i]);
       }
       int  end=sum;
       int mid;
       while(start<=end){
           mid=start+(end-start)/2;
           int count=0;
           int i=0;
           while(count<M){
               sum=0;
               while((sum+A[i])<=mid){
                   sum+=A[i];
                   i++;
               }
               count++;
           }
           if(i<=N-1){
               start=mid+1;
           }
           else{
               end=mid-1;
               res=mid;
           }
       }
       return res;
   }
};
```

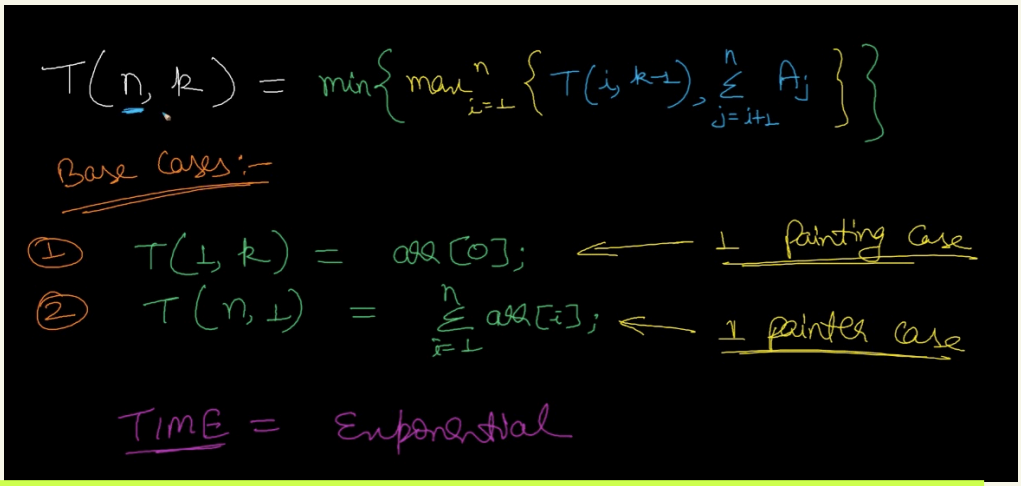








For each possible painter, we can partition at n different locations, therefore we need a for loop. After doing partition, take sum of right sub array and then we are dividing the array into two parts, calling this partition call with left sub array
I is array size and k-1 is number of painters. This backtracking approach will take exponential time.



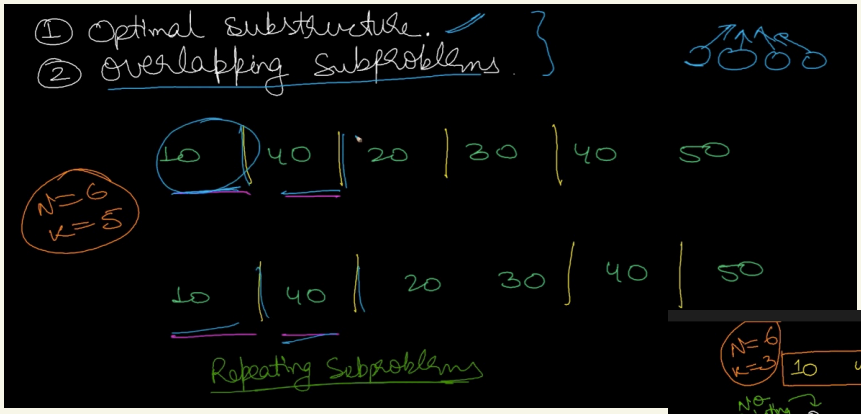
N paintings and K painters, make partition And find sum of right sub array, and the left sub array is passed to partition call with Decreased n value, and k all also decreased by one as one painter is assigned to the right hand side.

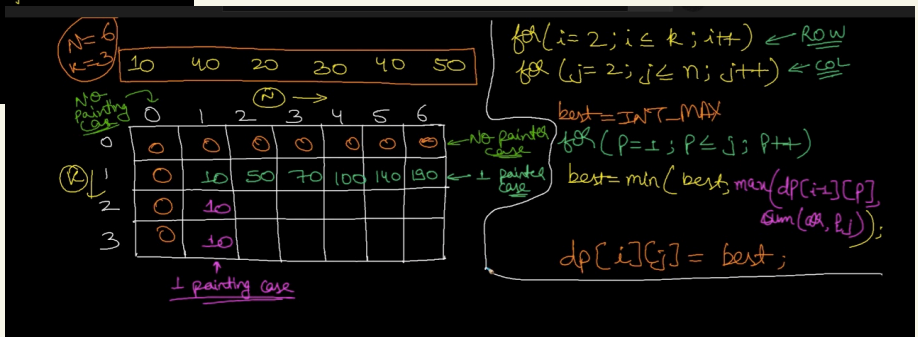
For this configuration, we will take maximum sum subarray value, do this for all possible configuration and take the minimum of the maximum sum subarray value, which will give optimal configuration which we should follow in order to decide the n paintings in k painters.

Base case 2: IF you have n paintings and 1 painters, total time is sum of all the array Elements. We want to improve this backtracking approach.
Before DP, find optimal substructure and overlapping subproblems.

Base cases: when you have one painting, no matter how many painters are there your painting time will be same

Optimal substructure: If we can solve a smaller sub problem, it will help to solve a bigger problem. We are calling partition on smaller subarray, and using the already solved partition, we can solve the entire array.
Overlapping Subproblem: 6 paintings and 5 painters





Binary search approach: O(n log n) method

```
class Solution
{
    bool isValid(int arr[], int n, int m, long long int k)
    {
        long int ptr=1;
        long long int sum=0;
        for(long int i=0;i<n ;i++)
        {
            sum+=arr[i];
            if(sum>k)
            {
                ptr++;
                sum=arr[i];
            }
            if(ptr>m) return false;
        }
        return true;
    }
    public:
    long long minTime(int arr[], int n, int m)
    {
        long long int i=0,j=0,res;
        for(long int k = 0; k < n ; k++)
        {
            if(arr[k] > i)
            {
                i = arr[k];
            }
            j += arr[k];
        }
        while(i <= j)
        {
            long long int mid=i + (j-i)/2;
            if(isValid(arr,n,m,mid) == true)
            {
                res = mid;
                j = mid - 1;
            }
            else
            {
                i = mid + 1;
            }
        }
        return res;
    }
};
```

10. Regular Expression Matching

Hard 7399 1030 Add to List Share

Given an input string `s` and a pattern `p`, implement regular expression matching with support for `'.'` and `'*'` where:

- `'.'` Matches any single character.
- `'*'` Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input: s = "aa", p = "a*"
Output: true
Explanation: '*' means zero or more of the preceding
element, 'a'. Therefore, by repeating 'a' once, it
becomes "aa".
```

Example 3:

```
Input: s = "ab", p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any
character (.)".
```

```
class Solution
{
public:
    bool isMatch(string s, string p)
    {
        int slen = s.length(), plen = p.length();
        vector<vector<bool>> dp(slen + 1, vector<bool>(plen + 1, false));
        dp[0][0] = true;
        // Note that s = "", can match p = "a*b*c*", hence `i` should be start with 0
        for (int i = 0; i <= slen; ++i)
        {
            for (int j = 1; j <= plen; ++j)
            {
                /* the index of '*' in `p`, must be >= 1, otherwise `p` is invalid,
                 * i.e. (j - 1 >= 1) ==> (j >= 2)
                 */
                if (p[j - 1] == '*')
                {
                    dp[i][j] = dp[i][j -  2] ||
                               (i > 0 && dp[i - 1][j] && (s[i - 1] == p[j - 2] || p[j - 2] == '.'));
                }
                else if (i > 0 && (p[j - 1] == '.' || s[i - 1] == p[j - 1]))
                {
                    dp[i][j] = dp[i - 1][j - 1];
                }

            }
        }
        return dp[slen][plen];
    }
};
```

Leetcode: 10. Regular Expression Matching.

The description of this problem is clear:

- Given an input string `s` and a pattern `p`, implement regular expression matching with support for `'.'` and `'*'`.
  - `'.'` matches any single character.
  - `'*'` matches zero or more of the preceding element.
  - It is guaranteed for each appearance of the character `'*'`, there will be a previous valid character to match.
- The matching should cover the **entire** input string (not partial).

Dynamic Programming

A popular solution is using dynamic programming.

Let `dp[i, j]` be true if `s[1 ... i]` matches `p[1 ... j]`. And the state equations will be:

- `dp[i, j] = dp[i-1, j-1]` if `p[j] == '.' || p[j] == s[i]`
- `dp[i, j] = dp[i, j-2]` if `p[j] == '*'`, which means that we skip `"x*"` and matching nothing (or match `x` for zero times).
- `dp[i, j] = dp[i-1, j] && (s[i] == p[j-1] || p[j-1] == '.')`, if `p[j] == '*'`, which means that `"x*"` repeats >=1 times, and `x` can be character or a dot `'.'`.

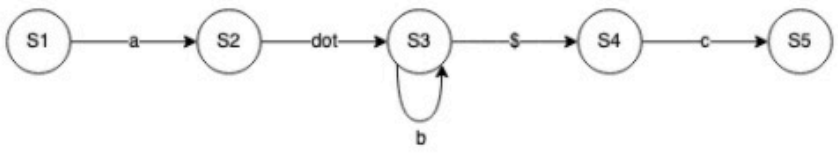
Please note that the index of `s` and `p` in the program start with `0`, which is a little different from the state equations above.

DFA

Here I want to show how can solve this problem by **deterministic finite automaton (DFA)**.

Please note that you should know the conception of DFA before you read this article.

For a pattern `p = "a.b*c"`, we can construct such a DFA:



The `'$'` sign means that, we do not need any character to reach next state, since `'*'` matches **zero** or more of the preceding element.

DFA can be represented by a graph in the program.

- `state` points to the last valid state we can reach.
- `state = 0` denote the invalid state, since `0` is the default value of `unordered_map`. Of course, we can use `-1`.

At last, we can also pass this problem by STL library :-D.

```
class Solution {
public:
    bool isMatch(string s, string p) {
        return regex_match(s, regex(p));
    }
};
```

```
class Solution
{
public:
    unordered_map<int, unordered_map<char, int>> automaton;
    int state = 1;
    bool isMatch(string s, string p)
    {
        int plen = p.length();
        for (int i = 0; i < plen; ++i)
        {
            char x = p[i];
            if ('a' <= x && x <= 'z' || x == '.')
                automaton[state][x] = state + 1, state += 1;
            else if (x == '*' && i > 0)
            {
                automaton[state - 1][p[i - 1]] = state - 1; // match >= 1 p[i - 1]
                automaton[state - 1]['$'] = state;          // match zero p[i - 1]
            }
        }
        return match(s, 0, 1);
    }

    /* idx - we are matching s[idx]
     * cur - current state in automaton
     */
    bool match(string &s, int idx, int cur)
    {
        int n = s.length();

        if (cur == 0) return false;
        if (idx >= n && cur == state) return true;

        if (idx < n)
        {
            /* Each node in automaton, has no more than 3 edges,
             * '$' means matching no character, hence still use `idx` in next matching.
             * For example, p = "a.b*c", s = "aac", we should output true,
             * match(s, idx, s3) means that, we skip matching "b*" in automaton (matched zero 'b').
             */
            int s1 = automaton[cur][s[idx]];
            int s2 = automaton[cur]['.'];
            int s3 = automaton[cur]['$'];
            return match(s, idx + 1, s1) || match(s, idx + 1, s2) || match(s, idx, s3);
        }
        else if (idx == n)
        {
            /* May be the last edge of automaton is `state[n-1] -- $ --> state[n]`
             * e.g. s = "aa", p = "a*"
             */
            return match(s, idx, automaton[cur]['$']);
        }
        return false;
    }
};
```

```
"aab"
"c*a*b"
```

here first character of both string `c!=a` why is expected answer: true ??
can u explain what is question asking to be done ?

The part "c*" means that it will match zero or some times for `c`. And the remained part "a*b" will match "aab", hence the answer is true.

▲ 2 ▼ Reply