DP editorial

Longest common subsequence

```
int Solution::solve(string A, string B) {
    int n=A.size();
    int m=B.size();
    vector<vector<int>> dp(n+1,vector<int>(m+1,0));
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(A[i-1]==B[j-1]){
                dp[i][j]=max(dp[i][j],dp[i-1][j-1]+1);
            }
            dp[i][j]=max({dp[i][j],dp[i-1][j],dp[i][j-1]});
        }
    }
    return dp[n][m];
}
```

Longest Palindromic Subsequence

Reverse one string and find longest common subsequence between it and original string

```
int Solution::solve(string a) {
    int n=a.size();
    string b=a;
    reverse(b.begin(),b.end());
    //cout<<b<<endl;
    vector<vector<int>> dp(n+1,vector<int>(n+1,0));
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            if(a[i-1]==b[j-1]){
                dp[i][j]=max(dp[i][j],dp[i-1][j-1]+1);
            }
            dp[i][j]=max({dp[i][j],dp[i-1][j],dp[i][j-1]});
            //cout<<dp[i][j]<<endl;
        }
    }
    return dp[n][n];

}
```

method 2

```
int Solution::solve(string a) {
    int n=a.size();
    string b=a;
    reverse(b.begin(),b.end());
    //cout<<b<<endl;
    vector<vector<int>> dp(n+1,vector<int>(n+1,0));
    for(int i=0;i<n;i++){
        dp[i][i]=1;
    }
    int ans=1;
    for(int l=1;l<n;l++){
        for(int i=0;i+l<n;i++){
            int r=i+l;
            if(l==1){
                if(a[i]==a[r]){
                    dp[i][r]=2;

                }
                else{
                    dp[i][r]=1;

                }
                ans=max(ans,dp[i][r]);
                continue;
            }
            if(a[i]==a[r]){
                dp[i][r]=dp[i+1][r-1]+2;
            }
            dp[i][r]=max({dp[i][r],dp[i+1][r],dp[i][r-1]});
            ans=max(ans,dp[i][r]);
        }
    }
    return ans;

}
```

Knapsack Type 1

```
ll dp[101][100001];//max sum with till prefix i with j weight

int main()
{
ios_base::sync_with_stdio(0);
cin.tie(0);
    ll n,w;
    cin>>n>>w;
```

```
    ll w1[n+1];

    ll a[n+1];
    for(int i=1;i<=n;i++){
        cin>>w1[i]>>a[i];
    }



    for(int i=1;i<=n;i++){
        for(int j=1;j<=w;j++){
           dp[i][j]=dp[i-1][j];
                if(j-w1[i]>=0){

                    dp[i][j]=max(dp[i-1][j-w1[i]]+a[i],dp[i][j]);
                    //cout<<dp[i][j]<<endl;




        }
    }
    }
ll ans=0;
    for(int i=0;i<=w;i++){
        ans=max(dp[n][i],ans);
    }
    cout<<ans<<endl;


    return 0;
}
```

Knapsack type2

```
ll dp[101][100001];



int main()
{
ios_base::sync_with_stdio(0);
cin.tie(0);
    ll n,w;
    cin>>n>>w;
    ll w1[n+1];

    ll a[n+1];

    //cout<<dp[0][0]<<endl;
```

```
    for(int i=1;i<=n;i++){
        cin>>w1[i]>>a[i];
    }
     //min weight to reach this value form 1...i items
    for(int i=0;i<=n;i++){
        for(int j=0;j<=100000;j++){
            dp[i][j]=mod;
            if(j==0){
                dp[i][j]=0;
            }
        }
    }
    //cout<<dp[0][0]<<endl;


    for(int i=1;i<=n;i++){
        for(ll j=1;j<=100000;j++){
            dp[i][j]=dp[i-1][j];
            if(j-a[i]>=0){
                dp[i][j]=min(dp[i][j],w1[i]+dp[i-1][j-a[i]]);
            }
        }




    }

    int ans=0;
    for(int i=1;i<=n;i++){
        for(int j=0;j<=100000;j++){

                if(dp[i][j]<=w){
                    ans=max(ans,j);
                }

        }
    }
    cout<<ans<<endl;



    return 0;
}
```

Minimum edit distance

```


int Solution::minDistance(string A, string B) {
    int n=A.size();
    int m=B.size();
    long long dp[n+1][m+1];
```

```
    for(int i=0;i<=n;i++){
        for(int j=0;j<=m;j++){
            dp[i][j]=1e9;
        }
    }
    dp[0][0]=0;
    dp[1][0]=1;
    dp[0][1]=1;
    for(int i=1;i<=n;i++){
        dp[i][0]=i;
    }
    for(int i=1;i<=m;i++){
        dp[0][i]=i;
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(A[i-1]==B[j-1]){
                dp[i][j]=dp[i-1][j-1];
            }
            else{
                 dp[i][j]=dp[i-1][j-1]+1;
            }
            dp[i][j]=min({dp[i][j],dp[i-1][j]+1,dp[i][j-1]+1});
        }
    }
    return dp[n][m];
```

Scramble String

```
int  dp[101][101][101];
int A[26];
string s1,s2;
int solve2(int i,int j,int s){
    if(dp[i][j][s]!=-1){
        return dp[i][j][s];
    }
    if(s==1){
        if(s1[i]==s2[j]){
            return dp[i][j][s]=1;
        }
        else{
            return dp[i][j][s]=0;
        }
    }
    string a=s1.substr(i,s);
    string b=s2.substr(j,s);
    sort(a.begin(),a.end());
    sort(b.begin(),b.end());
    if(a!=b){
```

```
        dp[i][j][s]=0;
        return dp[i][j][s];
    }
    for(int k=1;k<s;k++){
        if(solve2(i,j,k)&&solve2(i+k,j+k,s-k))return dp[i][j][s]=1;
        if(solve2(i,j+s-k,k)&&solve2(i+k,j,s-k))return dp[i][j][s]=1;
    }
    return dp[i][j][s]=0;







}
//string a,b;

int solve(string a,string b){
    if(a.size()!=b.size())return 0;
    string a1=a;
    string b1=b;
    int n=a.size();
    if(n==0)return 1;
    if(a==b)return 1;
    sort(a1.begin(),a1.end());
    sort(b1.begin(),b1.end());
    if(a1!=b1)return 0;
    for(int i=1;i<n;i++){
        // string s1=a.substr(0,i);
        // string s2=b.substr(0,i);
        // string s3=a.substr(i,n-i);
        // string s4=b.substr(i,n-i);
        if(solve(a.substr(0,i),b.substr(0,i))&&solve(a.substr(i,n-i),b.substr(i,n-
i))){
            return 1;
        }

        if(solve(a.substr(0,i),b.substr(n-i,i))&& solve(a.substr(i,n-
i),b.substr(0,n-i))){
            return 1;
        }




    }
    return 0;






}
int Solution::isScramble(const string A, const string B) {
    s1=A;
```

```
     s2=B;
     for(int i=0;i<51;i++){
         for(int j=0;j<51;j++){
             for(int k=0;k<51;k++){
                 dp[i][j][k]=-1;
             }
         }
     }
     if(A.size()!=B.size())return 0;
     return solve2(0,0,A.size());
    ///return solve(A,B);



}
```

Regular Expression Match

Implement wildcard pattern matching with support for '?' and '*' for strings A and B.

'?' : Matches any single character. '*' : Matches any sequence of characters (including the empty sequence). The matching should cover the entire input string (not partial).

```

int Solution::isMatch(const string A, const string B) {
   int n=A.size();
   int m=B.size();
   int c=0;
   for(int j=0;j<m;j++){
      if(B[j]!='*')c++;
   }
   if(c>n)return 0;
   if(c==0 && m>0)return 1;
   int dp[n+1][m+1];
   for(int i=0;i<n;i++){
      for(int j=0;j<m;j++)dp[i][j]=0;
   }
   dp[0][0]=1;
   for(int j=1;j<=m;j++){
      if(B[j-1]=='*'){
         dp[0][j]=dp[0][j-1];
      }
   }
   for(int i=1;i<=n;i++){
      for(int j=1;j<=m;j++){
         if(B[j-1]=='*'){
            dp[i][j]=dp[i][j-1]|dp[i-1][j];
```

```
            }
            else if(B[j-1]=='?'|| A[i-1]==B[j-1]){
               dp[i][j]=dp[i-1][j-1];
            }
            else{
               dp[i][j]=0;
            }
            //cout<<dp[i][j]<<endl;
         }
      }
      return dp[n][m];

   }
```

Regular Expression II

Implement regular expression matching with support for '.' and '*'.

'.' Matches any single character.

'*' Matches zero or more of the preceding element.

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

```
int Solution::isMatch(const string A, const string B) {
    string s=A;
    string p=B;
    int n=p.size();
    int m=s.size();
    s="0"+s;
    p="0"+p;

    int dp[n+1][m+1];
    memset(dp,0,sizeof(dp));
    dp[0][0]=1;

    for(int i=1;i<=n;i++){
        for(int j=0;j<=m;j++){
            if(p[i]==s[j]||p[i]=='.'){
                if(j>0)    dp[i][j]|=dp[i-1][j-1];
            }else if(p[i]=='*'){
                dp[i][j]|=dp[i-2][j];   //use 0 times
                dp[i][j]|=dp[i-1][j];   //use 1 times
                if(s[j]==p[i-1]||p[i-1]=='.')   if(j>0)    dp[i][j]|=dp[i][j-1];
//use multiple times
            }
        }
    }

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

```
}
```

Length of Longest Subsequence

Given an 1D integer array A of length N, find the length of longest subsequence which is first increasing then decreasing.

```
int calc(vector<int> A,int n,vector<int> &dp){
    dp.resize(n);
    dp[0]=1;
    int m[n];
    m[1]=A[0];
    int len=1;
    for(int i=1;i<n;i++){
        if(A[i]>m[len]){
            len++;
            dp[i]=len;
            m[len]=A[i];

        }
        else{
            int p=upper_bound(m+1,m+len+1,A[i])-m;
            if(m[p-1]==A[i]){
                p--;
            }
            dp[i]=p;
            m[p]=A[i];
        }
    }

    return 1;
}
int calc2(vector<int> A,int n,vector<int> &dp){
    dp.resize(n);
    dp[n-1]=1;
    for(int i=n-2;i>=0;i--){
        dp[i]=1;
        for(int j=i+1;j<n;j++){
            //dp[i]=1;
            if(A[i]>A[j]){
                dp[i]=max(dp[i],dp[j]+1);
            }
        }
    }
    return 1;
}

int Solution::longestSubsequenceLength(const vector<int> &A) {
```

```
    int n=A.size();
    if(n==0)return 0;
    //if(n==0)return 0;
    vector<int> d1;
    vector<int> d2;
    calc(A,n,d1);
    //reverse(A.begin(),A.end());
    calc2(A,n,d2);
    int ans=0;
    for(int i=0;i<n;i++){
        ans=max(ans,d1[i]+d2[i]-1);

    }
    //if(ans==1)return 0;
    return ans;
}
```

Method 2

```


class Solution {
public:
    int lengthOfLIS(vector<int>& a) {
        int n=a.size();
        vector<int> v;
        for(int i=0;i<n;i++){
            auto x=lower_bound(v.begin(),v.end(),a[i]);
            if(x==v.end()){
                v.push_back(a[i]);
            }
            else{
               int j=x-v.begin();

                v[j]=a[i];
            }
        }
        return v.size();


    }
};
```

Smallest sequence with given Primes

```


#define pb push_back
int a,b,c,d;
```

```
vector<int> Solution::solve(int A, int B, int C, int D) {
    a=A;
    b=B;
    c=C;
    d=D;
    vector<int> v;
    set<int> s;
    s.insert(A);
    s.insert(B);
    s.insert(C);
    while(v.size()!=D){
        auto x=s.begin();
        int p=*x;
        s.erase(x);
        s.insert(p*A);
        s.insert(p*B);
        s.insert(p*C);
        v.pb(p);
    }
    return v;

}
```

Tiling With Dominoes

Given an integer A you have to find the number of ways to fill a 3 x A board with 2 x 1 dominoes.

Return the answer modulo 109 + 7 .

```
int Solution::solve(int A) {
    int n=A;
    int mod=1000000007;
   int a[n+1],b[n+1];
   a[0]=1;
   a[1]=0;
   b[0]=0;
   b[1]=1;
   for(int i=2;i<=n;i++){
       a[i]=a[i-2]%mod+2*b[i-1]%mod;
       b[i]=a[i-1]%mod+b[i-2]%mod;
       a[i]%=mod;
       b[i]%=mod;


   }
   return a[n]%mod;
```

```
}
```

Ways to Decode

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

'A' -> 1 'B' -> 2 ... 'Z' -> 26 Given an encoded message A containing digits, determine the total number of ways to decode it modulo 109 + 7.

```
int Solution::numDecodings(string A) {
    int n=A.size();
    if(n==1){
        if(A[0]=='0')return 0;
        return 1;
    }
    int mod=1000000007;
    long long dp[n+1];
    memset(dp,0,sizeof(dp));
    if(A[0]=='0'&& A[1]=='0'){
        dp[0]=0;
    }
    else{
    dp[0]=1;
    }
    if(A[0]!='0')
    dp[1]=1;
    else
    dp[1]=0;
    for(int i=2;i<=n;i++){

        //cout<<p<<endl;
        if(A[i-2]=='1'|| (A[i-2]=='2'&& A[i-1]<'7')){
            dp[i]+=(dp[i-2])%mod;
            dp[i]%=mod;
        }
        if(A[i-1]!='0'){
        dp[i]+=dp[i-1]%mod;
        //cout<<dp[i]<<endl;
        dp[i]%=mod;
        }


    }
    return dp[n];


}
```

Intersecting Chords in a Circle

Given a number A, return number of ways you can draw A chords in a circle with 2 x A points such that no 2 chords intersect.

Two ways are different if there exists a chord which is present in one way and not in other.

Return the answer modulo 109 + 7.

=> If we draw a chord from a point in S_1 to a point in S_2, it will surely intersect the chord we've just drawn.

```
int Solution::chordCnt(int A) {
    int mod=1000000007;
    int n=2*A;
    long long dp[n+1]={0};
    dp[0]=1;
    dp[2]=1;
    for(int i=4;i<=n;i+=2){
        for(int j=0;j<i;j+=2){
            dp[i]+=(dp[j]%mod*dp[i-j-2]%mod)%mod;
            dp[i]%=mod;
        }
    }
    return dp[n]%mod;


}
```

Jump game array(Greedy)

Given an array of non-negative integers, A, you are initially positioned at the 0th index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

```
int Solution::canJump(vector<int> &A) {
    int n=A.size();
    if(n==1)return 1;
    if(A[0]==0)return 0;
     int c=1;
     for(int i=n-2;i>=0;i--){
         if(A[i]>=c){
             c=1;
         }
```

```
            else{
                c++;
            }

        }
        if(c==1)return 1;
        return 0;



}
```

Min Jump Array

Given an array of non-negative integers, A, of length N, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Return the minimum number of jumps required to reach the last index.

If it is not possible to reach the last index, return -1.

```
int Solution::jump(vector<int> &A) {
     int n=A.size();
    if(n==1)return 0;
    if(A[0]==0)return -1;
    int dp[n];
    for(int i=0;i<n;i++){
        dp[i]=100000009;
    }
    dp[0]=0;
    for(int i=0;i<n;i++){
        if(dp[i]==100000009)continue;
        for(int j=i+1;j<min(n,i+A[i]+1);j++){
            dp[j]=min(dp[j],dp[i]+1);
        }
    }
    if(dp[n-1]==100000009)return -1;
    return dp[n-1];

}
```

Longest Arithmetic Progression

Method 1

```
int Solution::solve(const vector<int> &A) {
    vector<int> B;
    for(int i=0;i<A.size();i++){
        B.push_back(A[i]);
    }
     int ans = 2;
    int n = A.size();
    if (n <= 2)
        return n;

    vector<int> llap(n, 2);

    sort(B.begin(), B.end());

    for (int j = n - 2; j >= 0; j--)
    {
        int i = j - 1;
        int k = j + 1;
        while (i >= 0 && k < n)
        {
            if (B[i] + B[k] == 2 * B[j])
            {
                llap[j] = max(llap[k] + 1, llap[j]);
                ans = max(ans, llap[j]);
                i -= 1;
                k += 1;
            }
            else if (B[i] + B[k] < 2 * B[j])
                k += 1;
            else
                i -= 1;
        }
    }
    return ans;

}
```

Method 2

```
int lenghtOfLongestAP(int set[], int n)
{
    if (n <= 2)  return n;

    // Create a table and initialize all values as 2. The value of
    // L[i][j] stores LLAP with set[i] and set[j] as first two
```

```
    // elements of AP. Only valid entries are the entries where j>i
    int L[n][n];
    int llap = 2;  // Initialize the result

    // Fill entries in last column as 2. There will always be
    // two elements in AP with last number of set as second
    // element in AP
    for (int i = 0; i < n; i++)
        L[i][n-1] = 2;

    // Consider every element as second element of AP
    for (int j=n-2; j>=1; j--)
    {
        // Search for i and k for j
        int i = j-1, k = j+1;
        while (i >= 0 && k <= n-1)
        {
           if (set[i] + set[k] < 2*set[j])
               k++;

           // Before changing i, set L[i][j] as 2
           else if (set[i] + set[k] > 2*set[j])
           {   L[i][j] = 2, i--;   }

           else
           {
               // Found i and k for j, LLAP with i and j as first two
               // elements is equal to LLAP with j and k as first two
               // elements plus 1. L[j][k] must have been filled
               // before as we run the loop from right side
               L[i][j] = L[j][k] + 1;

               // Update overall LLAP, if needed
               llap = max(llap, L[i][j]);

               // Change i and k to fill more L[i][j] values for
               // current j
               i--; k++;
           }
        }

        // If the loop was stopped due to k becoming more than
        // n-1, set the remaining entities in column j as 2
        while (i >= 0)
        {
            L[i][j] = 2;
            i--;
        }
    }
    return llap;
}
```

Shortest common superstring

Given a set of strings, A of length N.

Return the length of smallest string which has all the strings in the set as substring.

Ans =>

Brute force Let's say we have only two strings say s1 and s2, the possible cases are:

They do not overlap [ans = len(s1) + len(s2)] They overlap partially [ans = len(s1)+len(s2)-len(max. overlapping part)] They overlap completely [ans = max(len(s1), len(s2)] What we can see here is we can easily combine two strings. In the brute force, we could take all the permutations of numbers [1 .. N], then combine the strings in that order. e.g: strings = [s1, s2, s3], order = [2,3,1] Steps are: [s1, s2,s3] –> [s2+s3, s1] –> [s1+s2+s3]. (Here addition of strings is according to the method described above.

I would advice you to completely digest that this will give the optimal solution whatever the case may be. Considering all the permutations is optimal but time consuming.

Dynamic programming

We have dynamic programming to our rescue in this case. You can see that there is a optimal substructure and overlapping subproblems in the brute force algorithm described above. Well if you can't already see, let me help you out. Example: Input = [s1, s2, s3, s4] Order 1 = [2,3,1,4] , Steps: [s2+s3, s1, s4] –> [s2+s3+s1, s4] –> [s1+s2+s3+s4] Order 2 = [1,3,2,4] , Steps: [s1+s3, s2, s4] –> [s1+s2+s3, s4] –> [s1+s2+s3+s4].

Do you see here that Order1 and Order2 both calculated the optimal solution for set of strings [s1, s2, s3] (Intermediate string s1+s2+s3 is the optimal solution for this set)

Hurrah! Time to think Dynamically.

Bitmasking in DP Well, this kind of DP formulations require a specific technique called Bitmasking. It is not the conventional type and in this case T(N) = CCNN + CN*(2^N) (Still better than O(N!) right).

Formulation: dp[i][mask] = Optimal solution for set of strings corresponding to 1's in the mask where the last added string to our solution is i-th string.

Recurrence: dp[i][mask] = min(dp[x][mask ^ (1«i)] where {mask | (1«x) = 1})

I recommend you reading about the Bitmask in DP if you still have the doubt.

Happy coding.

P.S.: For those feeling excited, you can try finding the string (not the length) once you complete this one.

```
int funct(string A,string B, string& str)
{
    int max=INT_MIN;
    int len1=A.size(),len2=B.size();
    for(int i=1;i<=min(len1,len2);i++)
    {
```

```
        if(A.compare(len1-i,i,B,0,i)==0)
        {
            max=i;
            str=A+B.substr(i);
        }
    }
    for(int i=1;i<=min(len1,len2);i++)
    {
        if(A.compare(0,i,B,len2-i,i)==0)
        {
            max=i;
            str=B+A.substr(i);
        }
    }
    return max;
}
int Solution::solve(vector<string> &A) {
    int n=A.size();
    while(n!=1)
    {
        int l,r,maxi=INT_MIN;
        string resstr;
        for(int i=0;i<n;i++)
        {
            for(int j=i+1;j<n;j++)
            {
                string str;
                int res=funct(A[i],A[j],str);
                if(maxi<res)
                {
                    maxi=res;
                    resstr=str;
                    l=i;
                    r=j;
                }
            }
        }
        n--;
        if(maxi==INT_MIN)
        A[0]+=A[n];
        else
        {
            A[l]=resstr;
            A[r]=A[n];
        }
    }
    return A[0].size();
}
```

Travelling Salesman Problem

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Ans=>

Let the given set of vertices be {1, 2, 3, 4,....n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

Now the question is how to get cost(i)? To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i. We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
const int n = 4;
// give appropriate maximum to avoid overflow
const int MAX = 1000000;

// dist[i][j] represents shortest distance to go from i to j
// this matrix can be calculated for any given graph using
// all-pair shortest path algorithms
int dist[n + 1][n + 1] = {
    { 0, 0, 0, 0, 0 },    { 0, 0, 10, 15, 20 },
    { 0, 10, 0, 25, 25 }, { 0, 15, 25, 0, 30 },
    { 0, 20, 25, 30, 0 },
};

// memoization for top down recursion
int memo[n + 1][1 << (n + 1)];

int fun(int i, int mask)
{
    // base case
    // if only ith bit and 1st bit is set in our mask,
    // it implies we have visited all other nodes already
    if (mask == ((1 << i) | 3))
        return dist[1][i];
    // memoization
    if (memo[i][mask] != 0)
        return memo[i][mask];

    int res = MAX; // result of this sub-problem

    // we have to travel all nodes j in mask and end the
    // path at ith node so for every node j in mask,
    // recursively calculate cost of travelling all nodes in
```

```
    // mask except i and then travel back from node j to
    // node i taking the shortest path take the minimum of
    // all possible j nodes

    for (int j = 1; j <= n; j++)
        if ((mask & (1 << j)) && j != i && j != 1)
            res = std::min(res, fun(j, mask & (~(1 << i)))
                                    + dist[j][i]);
    return memo[i][mask] = res;
}
// Driver program to test above logic
int main()
{
    int ans = MAX;
    for (int i = 1; i <= n; i++)
        // try to go from node 1 visiting all nodes in
        // between to i then return from i taking the
        // shortest route to 1
        ans = std::min(ans, fun(i, (1 << (n + 1)) - 1)
                                + dist[i][1]);

    printf("The cost of most efficient tour = %d", ans);

    return 0;
}
```

Ways to color a 3xN Board

Given a 3 x A board, find the number of ways to color it using at most 4 colors such that no 2 adjacent boxes have same color.

Diagonal neighbors are not treated as adjacent boxes.

Return the ways modulo 109 + 7 as the answer grows quickly.

```
const long long mod=1e9+7;
int Solution::solve(int A) {
    vector<vector<int>> v;
    long long dp[A+1][4][4][4];
    for(int i=0;i<=A;i++){
        for(int i1=0;i1<=3;i1++){
            for(int i2=0;i2<=3;i2++){
                for(int i3=0;i3<=3;i3++){
                    dp[i][i1][i2][i3]=0;
                }
            }

        }
```

```
    }
    for(int i=0;i<=3;i++){
        for(int j=0;j<=3;j++){
            if(i==j)continue;
            for(int k=0;k<=3;k++){
                if(j==k)continue;
                v.push_back({i,j,k});
                dp[1][i][j][k]=1;


            }
        }
    }
    for(int i=2;i<=A;i++){
        for(auto &j:v){
            //dp[i][j[0]][j[1]][j[2]]=0;
            for(auto &u:v){
                if(j[0]==u[0] || j[1]==u[1] || j[2]==u[2])continue;
                dp[i][j[0]][j[1]][j[2]]+=dp[i-1][u[0]][u[1]][u[2]]%mod;
                dp[i][j[0]][j[1]][j[2]]%=mod;

            }
        }
    }
    long long ans=0;
    for(auto &x:v){
        ans+=dp[A][x[0]][x[1]][x[2]]%mod;
        ans%=mod;
    }
    return ans%mod;


}
```

Best Time to Buy and Sell Stock atmost B times

Given an array of integers A of size N in which ith element is the price of the stock on day i.

You can complete atmost B transactions.

Find the maximum profit you can achieve

```
int Solution::solve(vector<int> &A, int B) {
    int n=A.size();
    B=min(B,n-1);
    int dp[n+1][n+1];// max cost selling stock i
    for(int i=0;i<=n;i++){
        dp[0][i]=0;
        dp[i][0]=0;
```

```
    }
    int aux[n+1];
    for(int i=0;i<=n;i++){
        aux[i]=INT_MIN;
    }
    aux[0]=-A[0];
    for(int i=1;i<n;i++){

        for(int j=1;j<=B;j++){
            dp[i][j]=dp[i-1][j];
            dp[i][j]=max(dp[i][j],aux[j-1]+A[i]);
            // for(int k=0;k<i;k++){
            //     dp[i][j]=max(dp[i][j],A[i]-A[k]+dp[k][j-1]);
            // }
            aux[j]=max(aux[j],dp[i][j]-A[i]);

        }
        aux[0]=max(aux[0],-A[i]);

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

Kth Manhattan Distance Neighbourhood

Given a matrix M of size nxm and an integer K, find the maximum element in the K manhattan distance neighbourhood for all elements in nxm matrix.

In other words, for every element M[i][j] find the maximum element M[p][q] such that abs(i-p)+abs(j-q) <= K.

Note: Expected time complexity is O(N * N *K)

```
vector<vector<int> > Solution::solve(int A, vector<vector<int> > &B){
    vector<vector<int>> dp=B;
    for(int k=1;k<=A;k++)
    {
        vector<vector<int>> h(B.size(),vector<int>(B[0].size()));
        for(int i=0;i<B.size();i++)
        {
            for(int j=0;j<B[0].size();j++)
            {
                h[i][j]=dp[i][j];
                if(i>0)
                    h[i][j]=max(h[i][j],dp[i-1][j]);
                if(i<B.size()-1)
                    h[i][j]=max(h[i][j],dp[i+1][j]);
                if(j>0)
                    h[i][j]=max(h[i][j],dp[i][j-1]);
```

```
                if(j<B[0].size()-1)
                    h[i][j]=max(h[i][j],dp[i][j+1]);
            }
        }
        dp=h;
    }
    return dp;
}

```

Egg drop problem/Cut Logs

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.

```


class Solution {
public:
    int superEggDrop(int k, int n) {

vector<vector<int>> dp(k+1,vector<int>(n+1,0));//min moves for i eggs and j floors

for(int i=1;i<=n;i++)
    dp[1][i]=i;//one egg

for(int i=1;i<=k;i++)
{
    dp[i][1]=1;//one floor
}

for(int i=2;i<=k;i++)//dp[i][j]
{
    for(int j=2;j<=n;j++)
    {
        int l= 1;
        int r = j;
        while(l+1<r){
            int mid = (l+r)/2;
            if(dp[i-1][mid-1]<dp[i][j-mid]) l = mid;//first dp is increasing as
for same number of eggs if floors increases drops needed will increased only or
remains same...second dp is decreasing so we need optimal point
            else r = mid;
        }
```

```
            int x1 = max(dp[i-1][l-1],dp[i][j-l]);
            int x2 = max(dp[i-1][r-1],dp[i][j-r]);
            dp[i][j] = 1 + min(x1,x2);
        }
    }
    return dp[k][n];

        }
    };
```

Word Combination

You are given a string of length n and a dictionary containing k words. In how many ways can you create the string using the words?

```


struct node{
    map<char,int> m;
    int end=0;
    int cnt=0;

};
 node a[2000005];
 int ptr=2;
class Trie{
    public:

    void init(){
        for(int i=0;i<ptr;i++){
            a[i].m.clear();
            a[i].cnt=0;
            a[i].end=0;
        }
        ptr=2;
    }

    void insert(string s){
        int cur=1;
        for(auto x:s){
            if(a[cur].m.find(x)==a[cur].m.end()){
                a[cur].m[x]=ptr++;
            }
            cur=a[cur].m[x];
            a[cur].cnt++;
        }
        a[cur].end=1;
    }
    bool search(string s){
        int cur=1;
```

```
        for(auto x:s){
            if(a[cur].m.find(x)==a[cur].m.end()){
                return false;
            }
            cur=a[cur].m[x];
        }
        return a[cur].end;
    }



};
int solve(){
    Trie t;

    string s;
    cin>>s;
    int n;
    cin>>n;
    for(int i=0;i<n;i++){
        string x;
        cin>>x;
        t.insert(x);

    }
    int dp[s.size()+1];
    mem0(dp);
    n=s.size();// important  step
    dp[n]=1;//number of words which starts from this index
    //if a word end at j then dp[i]=dp[j+1] because it will begin new word from
j+1 and we need combinations such
    //that it starts from 0 and end at n-1
    for(int i=n-1;i>=0;i--){
        char c=s[i]-'a';
        int cur=1;
        int ans=0;
        for(int j=i;j<n;j++){
            if(a[cur].m.find(s[j])==a[cur].m.end())break;
            cur=a[cur].m[s[j]];
            if(a[cur].end){
                ans+=dp[j+1];
                ans%=mod;
            }

        }
        dp[i]=ans;

    }
    cout<<dp[0]<<endl;
    return 0;

}
```

Required Substring

Your task is to calculate the number of strings of length n having a given pattern of length m as their substring. All strings consist of characters A–Z

```
vector<int> kmp(string &s){
    int n=s.size();
    vector<int> p(n);
    for(int i=1;i<n;i++){
        int j=p[i-1];
        while(j>0 && s[i]!=s[j]){
            j=p[j-1];
        }
        if(s[i]==s[j]){
            j++;
        }
        p[i]=j;
    }
    return p;
}
vector<int> pk;
int dp[1005][101];
int done[1005][101];
ll state(int i,string &k,int n,int c,string &s){
    //if(k.size() && k[0]=='B')debug(k);
    if(n-i < (s.size()-c))return 0;
    if((done[i][c])){
        return dp[i][c];
    }
    //debug(i);
    if(c==s.size()){
        done[i][c]=1;
        return dp[i][c]=power(26,(n-i))%mod;


    }
    //debug(k);
    if(i==n){
        if(c==s.size()){
        //debug(k);
        //debug(c);
        return 1;
        }
        return 0;

    }

    ll res=0;
    for(int j=0;j<26;j++){
```

```
            char c1=char(int('A')+j);
            k+=c1;
            //debug(k);
            string s1=s+'#'+k;
            //debug(s1);
            int t=c;
            while(1){//using kmp here
            if(s[t]==c1){
                t++;
                break;


            }
            else if(t){
                t=pk[t-1];

            }
            else{
                break;
            }
            }

            int maxa=t;

            //if(k[0]=='A')
            //cout<<maxa<<endl;
            res+=state(i+1,k,n,maxa,s);
            res%=mod;
            k.pop_back();

        }
        //debug(k);
        //debug(c);
        //debug(res);
        done[i][c]=1;
        return dp[i][c]=res;


    }

    int solve(){
        int n;
        cin>>n;
        string s;
        cin>>s;
        pk=kmp(s);
        mem0(done);
        string ans="";
        cout<<state(0,ans,n,0,s)<<endl;;

        return 0;

    }
```

Coin Change Problem

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

```
class Solution {
public:
    int ans=INT_MAX;
    int dp[13][10001];
    long long state(vector<int> &a,int i,int amount){
        //cout<<i<<" "<<amount<<endl;
        if(i==a.size()|| amount<=0){
            if(amount==0){
                return 0;
            }
            else{
                return INT_MAX;
            }
        }
        if(dp[i][amount]!=-1){
            return dp[i][amount];
        }
        long long ans=0;
        if(a[i]<=amount){
            ans=min(1+state(a,i,amount-a[i]),state(a,i+1,amount));

        }
        else{
            ans=state(a,i+1,amount);
        }
       return dp[i][amount]=ans;

    }
    int coinChange(vector<int>& a, int amount) {

        int n=a.size();
        memset(dp,-1,sizeof(dp));
        long long ans=state(a,0,amount);
        if(ans>=INT_MAX)return -1;
        return ans;


    }
```

```
};
```

Subset Sum

```
class Solution {
public:
    bool canPartition(vector<int>& a) {
              int n=a.size();
        int s=0;
        for(auto x:a){
            s+=x;

        }
        if(s%2==1)return 0;


        vector<vector<int>> dp(n+10,vector<int>(s+10));
         dp[0][0]=1;
        for(int i=1;i<=n;i++){
            for(int j=0;j<=s;j++){
                dp[i][j]=dp[i-1][j];
                if(j-a[i-1]>=0){
                    dp[i][j]=dp[i][j]|dp[i-1][j-a[i-1]];
                }
            }
        }
        s/=2;
        return dp[n][s];

    }
};
```

Min cost to cut a stick

Given an integer array cuts where cuts[i] denotes a position you should perform a cut at.

You should perform the cuts in order, you can change the order of the cuts as you wish.

The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation.

```
class Solution {
```

```
public:
    int minCost(int i, int j,vector<int>&cuts, vector<vector<int>>&dp){
        //base condition
        if(i>j)return 0;

        //check the cache
        if(dp[i][j]!=-1)return dp[i][j];
        int mini=1e7;
        for(int idx=i;idx<=j;idx++){
            int cost=cuts[j+1]-cuts[i-1]+
                minCost(i,idx-1,cuts,dp)+minCost(idx+1,j,cuts,dp);
            mini=min(mini,cost);
        }
        return dp[i][j]=mini;
    }
    int minCost(int n, vector<int>& cuts) {
        int sz=cuts.size();
        cuts.push_back(n);
        cuts.insert(cuts.begin(),0);
        sort(cuts.begin(),cuts.end());
        vector<vector<int>>dp(sz+1,vector<int>(sz+1,-1));
        return minCost(1,sz,cuts,dp);
    }
};
```

Palindromic Partitioning

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 the given string.

```
class Solution{
public:
    int isPal(string &s,int i,int j){
        while(i<=j){
            if(s[i]!=s[j])return 0;
            i++;
            j--;
        }
        return 1;
    }
    int dp[505][505];
    int state(int i,int j,string &s){
        if(i==j){
            return 0;
        }
        //string k="";
        if(dp[i][j]!=-1)return dp[i][j];
```

```
        if(isPal(s,i,j))return dp[i][j]=0;
        int ans=INT_MAX;
        for(int x=i;x<j;x++){
            int p=state(i,x,s)+state(x+1,j,s)+1;
            ans=min(p,ans);

        }
        return dp[i][j]=ans;

    }
    int palindromicPartition(string str)
    {
        int n=str.size();
        memset(dp,-1,sizeof(dp));
        return state(0,n-1,str);

        // code here
    }
};
```

Tower of Hanoi

The Tower of Hanoi game consists of three stacks (left, middle and right) and n round disks of different sizes. Initially, the left stack has all the disks, in increasing order of size from top to bottom.

The goal is to move all the disks to the right stack using the middle stack. On each move you can move the uppermost disk from a stack to another stack. In addition, it is not allowed to place a larger disk on a smaller disk.

The pattern here is :

- Shift 'n-1' disks from 'A' to 'B', using C.
- Shift last disk from 'A' to 'C'.
- Shift 'n-1' disks from 'B' to 'C', using A.

```


START
Procedure Hanoi(disk, source, dest, aux)

   IF disk == 1, THEN
      move disk from source to dest
   ELSE
      Hanoi(disk - 1, source, aux, dest)     // Step 1
      move disk from source to dest          // Step 2
      Hanoi(disk - 1, aux, dest, source)     // Step 3
   END IF
```

```
END Procedure
STOP
```