**Partition Equal Subset Sum**

Given an array **arr[]** of size **N**, check if it can be partitioned into two parts such that the sum of elements in both parts is the same.

**Input:** N = 4

arr = {1, 5, 11, 5}

**Output:** YES

**Explaination:**

The two parts are {1, 5, 5} and {11}.

**Input:** N = 3

arr = {1, 3, 5}

**Output:** NO

**Explaination:** This array can never be

partitioned into two such parts.

class Solution{

static int equalPartition(int N, int arr[])

{

// code here

int sum=0;

for(int i=0;i<N;i++){

sum+=arr[i];

}

if(sum%2!=0){

return 0;

}

int[][] dp =new int[N+1][(sum/2)+1];

return subsetSum(arr,sum/2,dp,N);

}

static int subsetSum(int[] arr,int tar,int[][] dp,int n){

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

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

if(j==0){

dp[i][j]=1;

}

else{

dp[i][j]=dp[i-1][j];

if(j>=arr[i-1] && dp[i-1][j]!=1){

dp[i][j]= dp[i-1][j-arr[i-1]];

}

}

}

}

return dp[n][tar];

}

}

**2.Unbounded KnapSack**

i) **Knapsack with Duplicate Items**

Given a set of **N** items, each with a weight and a value, represented by the array **w[]** and **val[]** respectively. Also, a knapsack with weight limit **W**.  
The task is to fill the knapsack in such a way that we can get the maximum profit. Return the maximum profit.  
Note: Each item can be taken any number of times.

**Input:** N = 2, W = 3

val[] = {1, 1}

wt[] = {2, 1}

**Output:** 3

**Explanation:**

1.Pick the 2nd element thrice.

2.Total profit = 1 + 1 + 1 = 3. Also the total

 weight = 1 + 1 + 1 = 3 which is <= W.

class Solution{

static int knapSack(int N, int W, int val[], int wt[])

{

// code here

int[][] dp=new int[N+1][W+1];

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

for(int j=1;j<=W;j++){

if(j>=wt[i-1]){

dp[i][j]=Math.max(dp[i-1][j],val[i-1]+dp[i][j-wt[i-1]]);

}

else{

dp[i][j]=dp[i-1][j];

}

}

}

return dp[N][W];

}

}

ii) **Rod Cutting  Similar to unbounded KnapSack**

Given a rod of length **N** inches and an array of prices, **price[]** that contains prices of all pieces of size smaller than **N**. Determine the maximum value obtainable by cutting up the rod and selling the pieces.

**Input:**

N = 8

Price[] = {1, 5, 8, 9, 10, 17, 17, 20}

**Output:**

22

**Explanation:**

The maximum obtainable value is 22 by

cutting in two pieces of lengths 2 and

6, i.e., 5+17=22.

class Solution{

public int cutRod(int price[], int n) {

//code here

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

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

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

if(j>=i){

dp[i][j]=Math.max(dp[i-1][j],price[i-1]+dp[i][j-i]);

}

else{

dp[i][j]=dp[i-1][j];

}

}

}

return dp[n][n];

}

}

iii) **Coin Change - Number of ways**

Subset Sum with unbounded knapsack variation

You have an infinite supply of coins, each having some value. Find out the number of ways to use the coins to sum-up to a certain required value.

**Input:**

value = 4

numberOfCoins = 3

coins[] = {1,2,3}

**Output:** 4

**Explanation:** We need to make the change

for value = 4. The denominations are

{1,2,3} Change for 4 can be made:

1+1+1+1

1+1+2

1+3

2+2

So, as it is evident, we can do this in

4 ways.

class Solution

{

//Function to find out the number of ways to use the coins to

//sum up to a certain required value.

public long numberOfWays(int coins[],int numberOfCoins,int value)

{

long[][] dp=new long[numberOfCoins+1][value+1];

dp[0][0]=1;

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

for(int j=0;j<=value;j++){

if(j>=coins[i-1]){

dp[i][j]=dp[i-1][j]+dp[i][j-coins[i-1]];

}

else{

dp[i][j]=dp[i-1][j];

}

}

}

return dp[numberOfCoins][value];

}

}

iv) **Number of Coins**

Given a value **V** and array **coins[]** of size **M**, the task is to make the change for **V** cents, given that you have an infinite supply of each of coins{coins1, coins2, ..., coinsm} valued coins. Find the minimum number of coins to make the change. If not possible to make change then return -1.

**Input**: V = 30, M = 3, coins[] = {25, 10, 5}

**Output:** 2

**Explanation**: Use one 25 cent coin

and one 5 cent coin

class Solution{

public int minCoins(int coins[], int M, int value)

{

// Your code goes here

int[][] dp=new int[M+1][value+1];

for(int i=0;i<dp[0].length;i++){

dp[0][i]=Integer.MAX\_VALUE-1; //max\_value-1 bcoz max\_value+1 becomes min value and gives wrong answer while processing

}

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(j>=coins[i-1]){

dp[i][j]=Math.min(dp[i-1][j],dp[i][j-coins[i-1]]+1);

}

else{

dp[i][j]=dp[i-1][j];

}

}

}

return dp[M][value]==Integer.MAX\_VALUE-1?-1:dp[M][value];

}

}

3.Longest Common Subsequence

**Recursive Code:**

int lcs(String A,String B,int l1,int l2){

if(l1<0 || l2<0){

return 0;

}

if(A.charAt(l1)==B.charAt(l2)){

return 1+lcs(A,B,l1-1,l2-1);

}

else{

return Math.max(lcs(A,B,l1,l2-1),lcs(A,B,l1-1,l2));

}

}

**Memoization Code:**

public int longestCommonSubsequence(String text1, String text2) {

int[][] dp=new int[text1.length()+1][text2.length()+1];

for(int i=0;i<dp.length;i++){

for(int j=0;j<dp[0].length;j++){

dp[i][j]=-1;

}

}

return lcsMem(text1,text2,text1.length()-1,text2.length()-1,dp);

}

int lcsMem(String A,String B,int l1,int l2,int[][] dp){

if(l1<0 || l2<0){

return 0;

}

if(dp[l1][l2]!=-1){

return dp[l1][l2];

}

if(A.charAt(l1)==B.charAt(l2)){

return dp[l1][l2]=1+lcs(A,B,l1-1,l2-1);

}

else{

return dp[l1][l2]=Math.max(lcs(A,B,l1,l2-1),lcs(A,B,l1-1,l2));

}

}

**Tabulation:**

public int longestCommonSubsequence(String text1, String text2) {

int[][] dp=new int[text1.length()+1][text2.length()+1];

lcsDp(text1,text2,dp);

return dp[dp.length-1][dp[0].length-1];

}

void lcsDp(String A,String B,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(A.charAt(i-1)==B.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

}

}

}

}

ii) **Longest Common Substring**

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

**Input:** S1 = "ABCDGH", S2 = "ACDGHR"

**Output:** 4

**Explanation**: The longest common substring

is "CDGH" which has length 4.

class Solution{

int longestCommonSubstr(String S1, String S2, int n, int m){

// code here

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

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

for(int j=1;j<=m;j++){

if(S1.charAt(i-1)==S2.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=0;

}

}

}

int max=0;

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

for(int j=1;j<=m;j++){

if(max<dp[i][j]){

max=dp[i][j];

}

}

}

return max;

}

}

**iii)Printing Longest Common Subsequence**

**LCS + below Code**

int i=text1.length(),j=text2.length();

String res="";

while(i>0 && j>0){

if(text1.charAt(i-1)==text2.charAt(j-1)){

res=String.valueOf(text1.charAt(i-1))+res;

i--;

j--;

}

else{

if(dp[i-1][j]>dp[i][j-1]){

i--;

}

else{

j--;

}

}

}

System.out.println(res);

**iv)Shortest Common Supersequence**

**S1:AGGTAB**

**S2:GXTXAYB**

**S3:AGGXTXAYB**

**Input:**

X = abcd, Y = xycd

**Output:** 6

**Explanation:** Shortest Common Supersequence

would be abxycd which is of length 6 and

has both the strings as its subsequences.

class Solution

{

//Function to find length of shortest common supersequence of two strings.

public static int shortestCommonSupersequence(String X,String Y,int m,int n)

{

//Your code here

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

int lcs=lcs(X,Y,m,n,dp);

return m+n-lcs;

}

public static int lcs(String X,String Y,int m,int n,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(X.charAt(i-1)==Y.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

}

}

}

return dp[m][n];

}

}

**V)Minimum number of deletions and insertions.**

**Input:** str1 = "heap", str2 = "pea"

**Output:** 3

**Explanation:** 2 deletions and 1 insertion

**p** and **h** deleted from **heap**. Then, **p** is

inserted at the beginning One thing to

note, though **p** was required yet it was

removed/deleted first from its position

and then it is inserted to some other

position. Thus, **p** contributes one to the

**deletion\_count** and one to the

**insertion\_count**.

Insertion+Deletion=l1-lcs+l2-lcs

class Solution

{

public int minOperations(String str1, String str2)

{

// Your code goes here

int m=str1.length(),n=str2.length();

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

int lcs=lcs(str1,str2,m,n,dp);

return m+n-(2\*lcs);

}

public static int lcs(String X,String Y,int m,int n,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(X.charAt(i-1)==Y.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

}

}

}

return dp[m][n];

}

}

**vi)Longest Palindromic Subsequence**

**Input:**

S = "bbabcbcab"

**Output:** 7

**Explanation**: Subsequence "babcbab" is the

longest subsequence which is also a palindrome.

Lps=lcs(S,reverse(S))

class Solution

{

public int longestPalinSubseq(String S)

{

//code here

String rev="";

for(int i=S.length()-1;i>=0;i--){

rev+=S.charAt(i);

}

int[][] dp=new int[S.length()+1][S.length()+1];

return lcs(S,rev,S.length(),S.length(),dp);

}

public static int lcs(String X,String Y,int m,int n,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(X.charAt(i-1)==Y.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

}

}

}

return dp[m][n];

}

}

**vii)Minimum number of deletions.**

Given a string 'str' of size ‘n’. The task is to remove or delete the minimum number of characters from the string so that the resultant string is a palindrome. Find the minimum numbers of characters we need to remove.  
**Note:** The order of characters should be maintained.

**Input:** n = 7,str = "aebcbda"

**Output:** 2

**Explanation**: We'll remove 'e' and

'd' and the string become "abcba".

class Solution

{

int minDeletions(String S, int n)

{

// code here

String rev="";

for(int i=S.length()-1;i>=0;i--){

rev+=S.charAt(i);

}

int[][] dp=new int[S.length()+1][S.length()+1];

return S.length()-lcs(S,rev,S.length(),S.length(),dp);

}

public static int lcs(String X,String Y,int m,int n,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(X.charAt(i-1)==Y.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

}

}

}

return dp[m][n];

}

}

**Shortest Common Supersequence**

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.

**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.

class Solution {

public String shortestCommonSupersequence(String str1, String str2) {

int m=str1.length(),n=str2.length();

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

lcs(str1,str2,dp);

String res="";

int i=m,j=n;

while(i>0 && j>0){

if(str1.charAt(i-1)==str2.charAt(j-1)){

res=str1.charAt(i-1)+res;

i--;

j--;

}

else{

if(dp[i-1][j]>dp[i][j-1]){

res=str1.charAt(i-1)+res;

i--;

}

else{

res=str2.charAt(j-1)+res;

j--;

}

}

}

if(i!=0){

res=str1.substring(0,i)+res;

}

else if(j!=0){

res=str2.substring(0,j)+res;

}

return res;

}

public static void lcs(String X,String Y,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(X.charAt(i-1)==Y.charAt(j-1)){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

}

}

}

}

}

**Longest Repeating Subsequence**

Given a string str, find the length of the longest repeating subsequence such that it can be found twice in the given string. The two identified subsequences A and B can use the same ith character from string str if and only if that ith character has different indices in A and B.

**Input:**

str = "axxxy"

**Output:** 2

**Explanation:**

The given array with indexes looks like

a x x x y

0 1 2 3 4

The longest subsequence is "xx".

It appears twice as explained below.

**subsequence A**

x x

0 1 <-- index of subsequence A

------

1 2 <-- index of str

**subsequence B**

x x

0 1 <-- index of subsequence B

------

2 3 <-- index of str

We are able to use character 'x'

(at index 2 in str) in both subsequences

as it appears on index 1 in subsequence A

and index 0 in subsequence B.

class Solution

{

public int LongestRepeatingSubsequence(String str)

{

// code here

int[][] dp=new int[str.length()+1][str.length()+1];

return (lcs(str,str,dp));

}

public int lcs(String s1,String s2,int[][] dp){

for(int i=1;i<dp.length;i++){

for(int j=1;j<dp[0].length;j++){

if(s1.charAt(i-1)==s2.charAt(j-1) && i!=j){

dp[i][j]=1+dp[i-1][j-1];

}

else{

dp[i][j]=Math.max(dp[i][j-1],dp[i-1][j]);

}

}

}

return dp[dp.length-1][dp[0].length-1];

}

}

**Matrix Chain Multiplication**

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 **ith** matrix has the dimensions **(arr[i-1] x arr[i])**.

**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.

Recursive Approach:

static int mcmRecur(int[] arr,int i,int j){

if(i>=j){

return 0;

}

int lnm=Integer.MAX\_VALUE;

for(int k=i;k<=j-1;k++){

int tempSum=mcmRecur(arr,i,k)+mcmRecur(arr,k+1,j)+arr[i-1]\*arr[k]\*arr[j];

if(tempSum<lnm){

lnm=tempSum;

}

}

return lnm;

}

Memoization:

class Solution{

static int matrixMultiplication(int N, int arr[])

{

// code here

int[][] dp=new int[N+1][N+1];

return mcmMemoiz(arr,1,N-1,dp);

}

static int mcmMemoiz(int[] arr,int i,int j,int[][] dp){

if(i>=j){

return 0;

}

int lnm=Integer.MAX\_VALUE;

int arrik=0,arrkj=0;

if(dp[i][j]!=0){

return dp[i][j];

}

for(int k=i;k<=j-1;k++){

int tempSum=mcmMemoiz(arr,i,k,dp)+mcmMemoiz(arr,k+1,j,dp)+arr[i-1]\*arr[k]\*arr[j];

if(tempSum<lnm){

lnm=tempSum;

}

}

dp[i][j]=lnm;

return lnm;

}

}

**Palindromic patitioning**

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.

**Input:** str = "ababbbabbababa"

**Output:** 3

**Explaination:** After 3 partitioning substrings

are "a", "babbbab", "b", "ababa".

class Solution{

static int palindromicPartition(String str)

{

// code here

int n=str.length();

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

return ppMemoiz(str,0,str.length()-1,dp);

}

static boolean isPalin(String s ,int i,int j){

while(i<j){

if(s.charAt(i)!=s.charAt(j)){

return false;

}

else{

i+=1;

j-=1;

}

}

return true;

}

static int ppRecur(String str,int i,int j){

if(i>=j){

return 0;

}

if(isPalin(str,i,j)==true){

return 0;

}

int pp=Integer.MAX\_VALUE;

for(int k=i;k<=j-1;k++){

int temp=ppRecur(str,i,k)+ppRecur(str,k+1,j)+1;

if(temp<pp){

pp=temp;

}

}

return pp;

}

static int ppMemoiz(String str,int i,int j,int[][] dp){

if(i>=j){

return 0;

}

if(isPalin(str,i,j)==true){

return 0;

}

if(dp[i][j]!=0){

return dp[i][j];

}

int pp=Integer.MAX\_VALUE;

for(int k=i;k<=j-1;k++){

int temp=ppMemoiz(str,i,k,dp)+ppMemoiz(str,k+1,j,dp)+1;

if(temp<pp){

pp=temp;

}

}

dp[i][j]=pp;

return pp;

}

}

**Boolean Parenthesization**

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.

**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)).

class Solution{

static int countWays(int N, String S){

// code here

int[][][] dp=new int[N+1][N+1][3];

for(int i=0;i<dp.length;i++){

for(int j=0;j<dp[0].length;j++){

for(int k=0;k<dp[0][0].length;k++){

dp[i][j][k]=-1;

}

}

}

return bpRec(S,0,N-1,true,dp);

}

static int bpRec(String s,int i,int j,boolean isTrue,int[][][] dp){

if(i>j){

return 0;

}

if(i==j){

if(isTrue==true){

if(s.charAt(i)=='T'){

return 1;

}

else{

return 0;

}

}

else{

if(s.charAt(i)=='F'){

return 1;

}

else{

return 0;

}

}

}

int x=isTrue?1:2;

if(dp[i][j][x]!=-1){

return dp[i][j][x];

}

int count=0;

for(int k=i+1;k<=j-1;k+=2){

int lt=dp[i][k-1][1] = bpRec(s,i,k-1,true,dp);

int lf=dp[i][k-1][2] = bpRec(s,i,k-1,false,dp);

int rt=dp[k+1][j][1] = bpRec(s,k+1,j,true,dp);

int rf=dp[k+1][j][2] = bpRec(s,k+1,j,false,dp);

if(s.charAt(k)=='&'){

if(isTrue==true){

count+=lt\*rt;

}

else{

count+=((lt\*rf)+(lf\*rt)+(lf\*rf));

}

}

else if(s.charAt(k)=='|'){

if(isTrue==true){

count+=((lt\*rt)+(lt\*rf)+(lf\*rt));

}

else{

count+=(lf\*rf);

}

}

else{

if(isTrue==true){

count+=((lt\*rf)+(lf\*rt));

}

else{

count+=((lt\*rt)+(lf\*rf));

}

}

}

dp[i][j][x]=count%1003;

return count%1003;

}

}