Recursion Assignment

20BD1A1290-Jakkanagari Harika

Program 1:

Print all possible strings of length k that can be formed from a set of n characters.

Solution:

import java.util.\*;

class Main{

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

String s[]=new String[n];

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

s[i]=sc.next();

}

int k=sc.nextInt();

method(s,n,k);

}

public static void method(String a[],int n,int k){

method1(a,"",n,k);

}

public static void method1(String a[],String ans,int n ,int k){

if(k==0){

System.out.println(ans);

return;

}

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

String nans=ans+a[i];

method1(a,nans,n,k-1);

}

}

}

Program 2:

Find the number of unique paths in a n X m grid, starting from [0,0] to [n-1,m-1], where movement is only allowed either 1 cell down or 1 cell right at a time.

Solution:

import java.util.\*;

class Main {

public static int noofuniquepaths(int m, int n)

{

if (m == 1 || n == 1)

return 1;

return noofuniquepaths(m - 1, n)

+ noofuniquepathss(m, n - 1);

}

public static void main(String args[])

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

int m=sc.nextInt();

System.out.println(noofuniquepaths(n,m));

}

}

Example 1:

Input: m = 3, n = 2

Output: 3

Program 3:  
The head of a singly linked-list is given. The list can be represented as: L0 → L1 → … → Ln - 1 → Ln Reorder the list to be on the following form: L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → … The values in the list's nodes may not be modified, only nodes themselves may be changed.

Solution:

class LinkedList {

static Node head;

static class Node {

int data;

Node next;

Node(int d)

{

data = d;

next = null;

}

}

void printlist(Node node)

{

if (node == null) {

return;

}

while (node != null) {

System.out.print(node.data + " -> ");

node = node.next;

}

}

Node reverselist(Node node)

{

Node prev = null, curr = node, next;

while (curr != null) {

next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

node = prev;

return node;

}

void rearrange(Node node)

{

Node slow = node, fast = slow.next;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

Node node1 = node;

Node node2 = slow.next;

slow.next = null;

node2 = reverselist(node2);

node = new Node(0);

Node curr = node;

while (node1 != null || node2 != null) {

if (node1 != null) {

curr.next = node1;

curr = curr.next;

node1 = node1.next;

}

if (node2 != null) {

curr.next = node2;

curr = curr.next;

node2 = node2.next;

}

}

node = node.next;

}

public static void main(String[] args)

{

LinkedList list = new LinkedList();

list.head = new Node(1);

list.head.next = new Node(2);

list.head.next.next = new Node(3);

list.head.next.next.next = new Node(4);

list.head.next.next.next.next = new Node(5);

list.printlist(head); // print original list

list.rearrange(head); // rearrange list as per ques

System.out.println("");

list.printlist(head); // print modified list

}

}

Problem 6:

Print sums of all subsets of a given set of size n.

Solution:

import java.util.\*;

class Sumofsubsets{

static void subsetSums(int[] arr, int l, int r, int sum)

{

if (l > r) {

System.out.print(sum + " ");

return;

}

subsetSums(arr, l + 1, r, sum + arr[l]);

subsetSums(arr, l + 1, r, sum);

}

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

int[] a =new int[n];

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

a[i]=sc.nextInt();

}

subsetSums(arr, 0, n - 1, 0);

}

}

Example 1:

Input:

arr[] = {2, 3}

Output: 0 2 3 5

Problem 7:

0/1 Knapsack Problem: We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible. The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Solution:

import java.util.\*;

class Knapsackprogram{

public static int knapSack(int W, int wt[], int val[], int n)

{

if (n == 0 || W == 0)

return 0;

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

else

return Math.max(val[n - 1]

+ knapSack(W - wt[n - 1], wt,

val, n - 1),

knapSack(W, wt, val, n - 1));

}

public static void main(String args[])

{

Scanner sc=new Scanner(System.in);

int pn=sc.nextInt();

int p[] = new int[pn];

int w[] = new int[pn];

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

p[i]=sc.nextInt();

}

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

w[i]=sc.nextInt();

}

int W =sc.nextInt();

System.out.println(knapSack(W, w, p, pn));

}

}

Example 1:

Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1} Output: 3

Program 8:

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

Solution:

import java.util.\*;

public class PalindromePartitions {

public static boolean method(String s, int i, int j) {

while (i<= j) {

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

return false;

}

i++;

j--;

}

return true;

}

public static void method1(List<List<String>> result, List<String> d, String s, int n, int index) {

if (index == n) {

result.add(new ArrayList<>(d));

return;

}

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

if (method(s, index, i)) {

dump.add(s.substring(index, i + 1));

method1(result, d, s, n, i + 1);

dump.remove(d.size() - 1);

}

}

}

public static List<List<String>> allPalindromePartition(String s) {

List<List<String>> result = new ArrayList<>();

List<String> dump = new ArrayList<>();

int n = s.length();

method1(result, dump, s, n, 0);

return result;

}

public static void main(String[] args) {

Scanner sc=new Scanner(System.in);

String s =sc.next();

List<List<String>> result = allPalindromePartition(s);

System.out.println(result);

}

}

Example 1:

Input: s = "bcc" Output: [["b", "c", "c"], ["b", "cc"]]