CS1290

Jose Lujan

ID:80572649

Final exam

**1- Stone Game**

**Recursive Definition:**

To solve this problem we will follow two different paths and this depends on which “if” statement the code will enter. As we said we only have two options the first path will be if “Alex” is the player who is starting the game, the second path will be if “Lee” is player one. Basically this is just to know who will start the game, this makes a difference in the sense that the player one can get an advantage over player two, of course this doesn’t mean that player one will win, we still need to do one more which is to add the all the stones from each player separately and compare them to see who is the winner.

**How to store solution to subproblems:**

In this case I we will just store the into a new variable for each player, so in this case we will have two new variable that will sum the points that each player has, at the end it will compare the sum of the two and if Alex’s total points are greater that lee’s points it will return True.

**Code:**

mport java.util.ArrayList;  
import java.util.Scanner;

public class Stones {  
   private ArrayList<Integer> arrayList = new ArrayList<Integer>();  
   private boolean player1=true;

   public stonegame() {

       arrayList.add(2);  
       arrayList.add(5);  
       arrayList.add(3);  
       arrayList.add(2);  
       System.out.println("Contents of arraylist ");  
       for (int i : arrayList) {  
           System.out.print(i + " ");  
       }  
       System.out.println();  
   }

   private int alex\_sum = 0, lee\_sum = 0;

   boolean getWinner() {  
       if (player1) {//for alex

           if (arrayList.get(0) > arrayList.get(arrayList.size() - 1)) {  
               alex\_sum += arrayList.get(0);// get first  
               arrayList.remove(0);  
           } else {  
               alex\_sum += arrayList.get(arrayList.size() - 1);// get last  
               arrayList.remove(arrayList.size() - 1);

           }

           player1 = false;  
       } else {//for lee  
           if (arrayList.get(0) > arrayList.get(arrayList.size() - 1)) {  
               lee\_sum += arrayList.get(0);// get first  
               arrayList.remove(0);  
           } else {  
               lee\_sum += arrayList.get(arrayList.size() - 1);// get last  
               arrayList.remove(arrayList.size() - 1);

           }  
           player1 = true;  
       }  
       if (arrayList.size() > 0) {  
           getWinner();// recursive call  
       }  
       return alex\_sum > lee\_sum;  
   }

   public static void main(String[] args) {  
       System.out.println(new Stones().getWinner());

   }  
}

**3 - Palindromic Substrings**

**Recursive Definition:**

If we know that the string of characters has a different index at the beginning and at the end, we now have two more substrings. We know that the string we get as an input cannot be greater that 1000 characters, so if its greater it can just throw an exception.

**How to store solution to subproblems:**

In this case we will use a 2d array to store the characters, I see it something similar to the edit distance problem. We will compare first the index at point [0] and the index at point [len-1].

**Code:**

int countPalindromicSeq(string str)  
{  
int len = str.length();  
  
  
int countPs[len+1][len+1]; // 2D array to store the count of palindromic subsequence  
memset(countPs, 0 ,sizeof(countPs));  
  
// all palindromic subsequence having length equals to 1  
for (int i=0; i<len; i++)  
countPs[i][i] = 1;  
  
// check if substring of length L is palindrome or not  
for (int L=2; L<=len; L++)  
{  
for (int i=0; i<len; i++)  
{  
int j = L+i-1;  
if (str[i] == str[j])  
countPS[i][j] = countPS[i][j-1] +  
countPS[i+1][j] + 1;  
else  
countPs[i][j] = countPS[i][j-1] +  
countPs[i+1][j] -  
countPs[i+1][j-1];  
}  
}  
  
return countPs[0][len-1];  
}

int main()  
{  
string str = "abaab";  
return 0;  
}

**3 – Arithmetic Slices**

**Recursive Definition:**

If the array length is less than three it will return 0 (a.lenght <3). We will use a for loop that will traverse the array, but i = 2 this is in the case that the array has a length of three (a.length = 3). If the array has 4 integer and all of them are different, we can spitted into three new arrays. In the first new array we will have all integers except for the last one(a.lenght-1), in the second new array we will have all the integers except that this time we will skip a[0] and the last new array will get filled will all the integers in the original array.

**How to store solution to subproblems:**

To store the solution to a subproblem we will use a new variable which will be “temp” and a new array that will store the values from temp each time it has been updated. At final the count array will tell us how many times we can slice the original array.

**Code:**

java recursive

public static int numberOfArithmeticSlices(int[] A) {

int[] count = new int[1];

int lastIndex = A.length - 1;

helper(A, lastIndex, count);

return count[0];

}

private static int helper(int[] A, int lastIndex, int[] count) {

if (lastIndex < 2) {

return 0;

}

int temp = 0;

if (A[lastIndex] - A[lastIndex - 1] == A[lastIndex - 1] - A[lastIndex - 2]) {

temp = 1 + helper(A, lastIndex - 1, count);

count[0] = count[0] + temp;

} else {

helper(A, lastIndex - 1, count);

}

return temp;

}

**6 - Maximum Length of Pair Chain**

**Recursive Definition:**

First we are given an input of several arrays, in which each array has two integers. We will compare (a[1],i-1) , and if the second item is the same as the previous one from the other array we will skip that number, so this will eliminate an array.

**How to store solution to subproblems:**

We could just store the values into a new array, but since we just want to have two integers per array, we will create a linked-list with with two items per node.

**Code:**

*def compare(item1,item2):*

*if bool(item1[1]-item2[1]):*

*return item1[1]-item2[1]*

*return item1[0]-item2[0]*

*def calc\_ans(lst,n):*

*if n==0:*

*return 0,lst[0][0]-1*

*ans,lim =calc\_ans(lst,n-1)*

*# If the starting point of current node > previous end point*

*# Return updated answer and end point*

*if lst[n-1][0]>lim:*

*return ans+1,lst[n-1][1]*

*return ans,lim*

**7 - Integer Break**

**Recursive Definition:**

We will have two edge cases, one in which n==0 and the second one in which n==1. This will return 1. After edge cases, if “n” == 10 we will have a for loop from (1,n-1).

**How to store solution to subproblems:**

To store the value of smaller problem, we will have a variable which has the max value, from that value we will subtract one. max= max-I;

**Code:**

def getMaxProductOfn(n):  
# Base cases  
if (n == 0 or n == 1):  
return 1  
max\_val = 1

**9 - Perfect Square**

**Recursive Definition:**

In this problem the main solution will come by getting the square root of “n “ which is the input given by the user. The result of that would be the max value, now we just need to compare it to the value of n if its greater we will use the square root of n + 1.

**Code:**

public int PerfectSquares(int n) {

int max = (int) Math.sqrt(n);

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

Arrays.fill(dp, Integer.MAX\_VALUE);

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

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

if(i==j\*j){

dp[i]=1; }

else if(i>j\*j){ dp[i]=Math.min(dp[i], dp[i-j\*j] + 1);

}

}

}

return dp[n];

}