**DSA PROJECT**

Course Code: IT200



**Submitted to:**

Ms. Chaitra Bhat

Assistant Lecturer

Department of Information Technology

NITK Surathkal

SACHIN – 15IT234

9th November 2016

**Contents:**

**Problem Number: 7**

**Gottacatch'em all!**

Pages :

* Problem Statement : 3 - 4
* Summary :5 - 6
* Code : 7 - 10
* Sample Outputs : 11

**Problem Number: 50**

**Mancunian And Colored Tree**

Pages :

* Problem Statement :12
* Summary :13 - 14
* Code : 15 - 17
* Sample Outputs :17

*(The document consists of* ***17*** *pages in total of which the code is embedded within after the summary of every Problem including the cover pages.)*

## Problem Number: 7

## Gotta catch 'em all!

Little Arihant has always wanted to be the best Pokémon trainer in this world. And he thinks he has achieved his goal, so he wants to quickly go and meet Professor Oak and verify this fact. But like all Pokémon trainers, he has a weird habit, too. He catches Pokémon which can go through evolution to become a better one. After roaming in the Pokeworld for days, he has finally managed to catch *k* such Pokémon.

The way he can make a Pokémon go through evolution is NOT by making them fight battles, but by using an *evolution stone.* Since he has *k* Pokémon, he naturally needs *k* evolution stones for every one of them, as well.

Now it takes little Arihant one complete day, to use the evolution stone on one Pokémon. And for each Pokémon, he knows how many days will they take to evolve after the evolution stone has been used on them.

He will go to meet Professor Oak, the very *next* day, once all his Pokémon have gone through evolution. He can select the order of applying evolution stones as he likes, so he wants to do it insuch a way that he gets to meet Professor Oak as soon as possible!

## Input

First line contains an integer k which denotes number of Pokémon.

Second line has k space separated integers each of which denotes the number of days for the ithPokémon to evolve.

## Output

The earliest day on which little Arihant can go and meet Professor Oak.

## *Sample Input Sample output Constraints*

**2 5 1 ≤ k ≤ 10^5**

**3 1 1 ≤days ≤ 10^6**

## Algorithm Used

**STEP 1:**

We take the array input a[] which stores the number of days required for each Pokémon to evolve.

**STEP 2:**

We sort the array in descending order( using merge sort) and initialize days to one.

**STEP 3:**

Start loop

Iterate through the array a[] and check if the value of a[i] + i is greater than the number of days.

If it is greater , update the value of days to a[i]+i

End loop

**STEP 4:**

Finally , increment the value of days since Arihant will go to the Professor on the day following the day on which all the Pokémon have evolved (which is the value of days after the loop).

**STEP 5:**

Print the value of “days” which is the minimum number of days required by Arihant to meet Professor Oak.

## Things To Note (Explanation of steps)

1. The array is sorted in descending order since the evolution stone should be first used on the Pokémon taking maximum number of days to evolve, so that the remaining evolution stones can be used on the rest of the Pokémon when the first one is evolving.
2. In STEP 2 , “i” is added to a[i] for checking since “i” denotes the

number of days already used up for using evolution stones. Thus adding a[i] to i gives the number of days consumed when the

ithPokémon evolves. If this value is greater than the already existing value of days, we update it with the new maximum value.

## Data Structure Used: ARRAY

Array is used in program because if we use other data structure like

using linked list then each time we have check for the required node sequentially by which time complexity will increase. An array offers

random access to its elements at any time. This makes sorting an array a lot easier than sorting a linked list. When traversing a linked list it needs to store values in separate nodes which are linked together by addresses.

Arrays take away this defect and allow us to directly input the data into a sequentially allotted memory location.

## Code

#include<stdio.h>

//This function merges two sorted arrays.

//The array is in Descending order.

/\*

Parameters Required:

1) arr[] = The final array.

2) l = The left limit or index.

3) m = The middle index.

4) r = The right limit or index.

\*/

void merge(intarr[],intl,intm,int r)

{

Inti=0,j=0,k=0;

int n1=m-l+1;//size of left array.

int n2=r-m;//size of right array.

int L[50000],R[50000];

/\*

(L[]) = temporary array to store left array.

(R[]) = temporary array to store right array.

(n1) = number of elements in the left array.

(n2) = number of elements in the right array.

(i) = index variable for left array.

(j) = index variable for right array.

(k) = index variable for merged array.

\*/

// Storing left array in (L[]).

for(i=0;i<n1;i++)

L[i]=arr[l+i];

// Storing left array in (R[]).

for(j=0;j<n2;j++)

R[j]=arr[m+1+j];

// Reset counters.

i=j=0; // Reset to 0.

k=l; // To start of the array.

// Transfer elements from (L[]) and (R[]) to (arr[]).

while(i<n1 && j<n2)

{

if(L[i]>=R[j]) // Transfer largest number first.

{

arr[k]=L[i];

i++;//increase index of (L[])

}

else

{

arr[k]=R[j];

j++;//increase index of (R[])

}

k++;//increase index of (arr[])

}

// Transfer all remaining elements from (L[])

while(i<n1)

{

arr[k]=L[i];

i++;

k++;

}

// Transfer all remaining elements from (R[])

while(j<n2)

{

arr[k]=R[j];

j++;

k++;

}

}

/\*

This function sorts the array in descending order using MergeSort.

Parameters Required:

1) arr[] = The array which needs to be sorted.

2) l = The lower limit.

3) r = The higher limit.

\*/

voidmergeSort(intarr[],intl,int r)

{

int m=0;// Middle element

if (l<r)

{

m=(l+r)/2;// The array is split in two arrays

// Funtion is called recursively to sort the left half of the array.

mergeSort(arr,l,m);

// Funtion is called recursively to sort the right half of the array.

mergeSort(arr,m+1,r);

// Calls merge function to merge the sorted left and right arrays.

merge(arr,l,m,r);

}

}

Intmain()

{

// This array stores the value of days required for the Pokemon to evolve.

Inta[100000];

Inti=0,k=0,days=0;

/\*

(i) = loop variable.

(k) = number of Pokemon.

(days) = number of days required to visit the Professor Oak.

\*/

// Input for Number of Pokemon. (k)

scanf("%d",&k);

// Input days required for evolution for (k) Pokemon. (a[])

for(i=0;i<k;i++)

scanf("%d",&a[i]);

// Sorts the array in descending order.

mergeSort(a,0,k-1);

// Algorithm to calculate number of days required to visit the Professor Oak.

days =1;

// Set days to 1 to apply the Evolution stone on the first Pokemon.

for(i=0;i<k;i++)

{

// Checks if days needed by the current Pokemon is more than the current days needed.

if(a[i]+i+1>days)

{

days=a[i]+i+1;

// if days needed is more change days to new value.

}

}

days++;// Increment to account for the day of the visit.

printf("%d",days);// Displays the answer.

return 0;

}

## TimeComplexity : O(n),where n= number of Pokémon

## Sample Output

|  |  |  |
| --- | --- | --- |
| TEST CASE | INPUT | OUTPUT |
| **1** | **9**  **1330 67 1212**  **420 1668 311**  **1725 92 1404** | **1727** |
| **2** | **4**  **8 2 4 7** | **10** |
| **3** | **6**  **11 9 15 2 10 4** | **17** |
| **4** | **5**  **101 54 98 43 2** | **103** |

## Problem Number: 50

# Mancunian and Colored Tree

After a hectic week at work, Mancunian and Liverbird decide to go on a fun weekend camping trip. As they were passing through a forest, they stumbled upon a unique tree of N nodes. Vertices are numbered from 1 to N.

Each node of the tree is assigned a color (out of C possible colors). Being bored, they decide to work together (for a change) and test their reasoning skills. The tree is rooted at vertex 1.

For each node, they want to find its closest ancestor having the same color.

## Input format

The first line contains two integers N and C denoting the number of vertices in the tree and the number of possible colors.

The second line contains N−1 integers. The I th integer denotes the parent of the i+1 th vertex.

The third line contains N integers, denoting the colors of the vertices. Each color lies between 1 and C inclusive.

## Output format

Print N space-separated integers. The ith integer is the vertex number of lowest ancestor of the I th node which has the same color. If there is no such ancestor, print −1for that node.

***SAMPLE INPUT SAMPLEOUTPUT CONSTRAINTS***

**5 4 -1 -1 -1 1 3 1≤N≤100,000**

**1 1 3 3 1≤C≤100,000**

**1 4 2 1 2**

## Explanation

Vertices 1, 2 and 3 do not have any ancestors having the same color as them. The nearest required ancestors for vertices 4 and 5 are vertices 1 and 3 respectively.

## DataStructure Used : ARRAY

Array is used in program because if we use other data structure like trees

using linked list then at each time we have check for the required node sequentially by which time complexity will increase.

If trees were used then there might be come errors or ambiguity, as the different nodes will have to be input in a specific order. For example, if node tree has to have node 5 as a parent. It would be difficult to create

That tree as node 5 has not yet been created. Arrays take away this ambiguity and allow us to directly input the parent.

## Algorithm Used

**STEP 1:**

We take the number of the nodes (n), numbers of colors(c) and the array (a[]) which stores the color of notes in it as input.

**STEP 2:**

**Start loop A:**

For each iteration:

Assign the value of parent of node (i) to a variable (say temp1).

For default ancestor or no ancestor initialize a variable (say temp2) = - 1

**Start loop B:**

Iterate the loop until the root is reached and checked for ancestor.

**If** color of ancestor is same as node color then Assign temp2 = temp 1 and

break the loop.

**If not**

Update the temp1 to parent of each node so in each node we move up the tree and finally trace each node.

**End loop B.**

Stores the Result of ancestor of each node in third row of input array.

**End loop A**.

**STEP 3:**

**Start loop:**

Print the ancestor of each node.

**End loop.**

## Code

#include<stdio.h>

intmain() //Main method.

{

// Variables needed.

Inta[3][100001]; // This array has three rows

/\*

Three rows are :

1. Parent of the node. (a[0][x])
2. Color of the node. (a[1][x])
3. Nearest ancestor of the node which has the same color as the node. (a[2][x])

\*/

int temp1=0,temp2=0,n=0,c=0,i=0;

/\*

(n) = Number of nodes.

(c) = Number of colors.

(i) = Loop variable.

(temp1) and (temp2) are temporary variables.

\*/

scanf("%d",&n); // Input number of nodes (n). scanf("%d",&c); // Input of number of colors (c).

// Input parents of nodes (2 to n).

// Parent of node 1 is 0.

// Since node 1 is considered the root.It has no parent.

a[0][1]=0;

for(i=2;i<=n;i++)

scanf("%d",&a[0][i]);

// Input colors of nodes (1 to n).

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

scanf("%d",&a[1][i]);

// Checking algorithm.

// Finds nearest ancestor which has the same color as the node (i).

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

{

// (temp1) is assigned the parent of node (i) every iteration.

temp1=a[0][i];

// (temp2) is assigned the value of (-1) every iteration.

// (-1) = Default ancestor (or) no ancestor.

temp2=-1;

// Loop is executed until the root is reached and checked.

// i.e when (temp1 = 0) implies root or node 1 has been reached.

for(;temp1!=0;)

{

// Checks if color of ancestor is the same as the node (i). if(a[1][i] == a[1][temp1])

{

// (temp2) is assgined the value of the ancestor. temp2=temp1; break;

// Loop is terminated since ancestor has been found.

}

// Update statement.

// (temp1) is updated to (the parent of the current node).

temp1=a[0][temp1];

// Each iteration (temp1) keeps moving up the tree.

}

// Result is stored in the third row of the array. (a[2][i]) a[2][i]=temp2;

}

// This loop displays the result.(a[2][i])

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

printf("%d ",a[2][i]);

return 0;

}

TimeComplexity : **O(n^2)** , **where n = number of nodes**

## SampleOutput

|  |  |  |
| --- | --- | --- |
| TEST CASE | INPUT | OUPUT |
| **1** | **5 3**  **1 1 2 3**  **1 2 3 1 2** | **-1 -1 -1 1 -1** |
| **2** | **5 4**  **1 2 2 1**  **4 2 1 4 3** | **-1 -1 -1 1 -1** |
| **3** | **7 3**  **1 1 2 3 3 5**  **3 2 1 2 3 2 2** | **-1 -1 -1 2 1 -1 -1** |

Thank You