**Sparse Arrays**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

int main() {

int N, Q;

char \*N\_array[1000], \*Q\_array[1000];

scanf("%d", &N);

for (int N\_i = 0; N\_i < N; N\_i++) {

char s[21];

scanf("%s", s);

N\_array[N\_i] = malloc(21);

strcpy(N\_array[N\_i], s);

}

scanf("%d", &Q);

for (int Q\_i = 0; Q\_i < Q; Q\_i++) {

int occurs = 0, result;

char s[21];

scanf("%s", s);

Q\_array[Q\_i] = malloc(21);

strcpy(Q\_array[Q\_i], s);

for (int N\_i2 = 0; N\_i2 < N; N\_i2++) {

result = strcmp(Q\_array[Q\_i], N\_array[N\_i2]);

if (result == 0) occurs++;

}

printf("%d\n", occurs);

}

return 0;

}

Array Manipulation

#include<stdio.h>

long A[10000009]={0},CF[10000009+1]={0};

int main()

{

  long N,Q;

    long val,left,right,i,count=0;

   long maxv=-1;

    scanf("%ld%ld",&N,&Q);

  for(i=0;i<Q;i++)

  {

   scanf("%ld%ld%ld",&left,&right,&val);

   CF[left-1]+=val;

   CF[right]-=val;

  }

  for(i=0;i<N;i++)

  {

   count+=CF[i];

   A[i]=count;

      if(count>maxv) maxv=count;

  }

       printf("%ld\n",maxv);

 return 0;

}

**Swap Nodes [Algo]**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

struct node {

    int data;

    struct node \*left;

    struct node \*right;

};

struct node\* create\_node(int val){

    if(val == -1){

        return NULL;

    }

    struct node \*temp=(struct node\*)malloc(sizeof(struct node));

    temp->data=val;

    temp->left=NULL;

    temp->right=NULL;

    return temp;

}

void inorder(struct node \*root){

    if(!root){

        return;

    }

    inorder(root->left);

    printf("%d ", root->data);

    inorder(root->right);

}

int max(int a, int b){

    if(a>b){

        return a;

    } else {

        return b;

    }

}

int height(struct node \* root){

    if(!root){

        return 0;

    }

    return(1+max(height(root->left),height(root->right)));

}

void swap\_nodes\_at\_level(struct node \*root, int inc, int level, int height){

    struct node \*tnode;

    if(!root){

        return;

    }

    if(level > height){

        return;

    }

    if(!(level%inc)){

        tnode=root->left;

        root->left=root->right;

        root->right=tnode;

    }

    swap\_nodes\_at\_level(root->left, inc, level+1, height);

    swap\_nodes\_at\_level(root->right, inc, level+1, height);

}

int tail=0;

int head=0;

void enqueue(struct node \*\*queue, struct node \*root){

    queue[tail]=root;

    tail++;

}

struct node\* dequeue(struct node \*\*queue){

    struct node \*temp = queue[head];

    head++;

    return temp;

}

int main() {

    /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/

    int nodes\_count, i, temp, h, tc\_num, index, inc, temp1, temp2;

    scanf("%d", &nodes\_count);

  //  printf("%d\n", nodes\_count);

    // int arr[2\*nodes\_count+1];

    struct node \*root\_perm, \*root\_temp;

    //queue=create\_queue(nodes\_count);

    struct node \*q[nodes\_count];

    for(i=0;i<nodes\_count;i++){

        q[i]=NULL;

    }

    //building the array

    //arr[0]=1;

   // for(i=1;i<=2\*nodes\_count;i++){

     //   scanf("%d",&temp);

      //  arr[i]=temp;

   //   printf("%d ", arr[i]);

   // }

    i=0,index=1;

    root\_temp=root\_perm=create\_node(1);

    enqueue(q, root\_temp);

    while(index<=2\*nodes\_count) {

        //printf("\n In Loop : i : %d",i);

        root\_temp=dequeue(q);

        //setting up the left child

        scanf("%d", &temp1);

        if(temp1 == -1){

        } else {

            root\_temp->left=create\_node(temp1);

            enqueue(q, root\_temp->left);

        }

        //setting up the right child

        scanf("%d", &temp2);

        if(temp2==-1) {

        } else {

            root\_temp->right=create\_node(temp2);

            enqueue(q, root\_temp->right);

        }

        index=index+2;

      //  i++;

    }

    h = height(root\_perm);

    scanf("%d", &tc\_num);

    //printf("%d",tc\_num);

    //printf("\n");

    //inorder(root\_perm);

    while(tc\_num){

        scanf("%d",&inc);

        temp=inc;

        //while(temp < height){

        swap\_nodes\_at\_level(root\_perm, inc, 1, h);

        //temp=temp + inc;

        //}

        //temp=0;

        inorder(root\_perm);

        printf("\n");

        tc\_num--;

    }

    //Tree is created at this point

    return 0;

}

Kitty's Calculations on a Tree

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define LIMIT 1000000007

typedef struct tree\_node\_list {

struct tree\_node \*node;

struct tree\_node\_list \*next;

} tree\_node\_list;

typedef struct tree\_node {

struct tree\_node \*parent;

uint32\_t num;

int32\_t depth;

} tree\_node;

typedef struct aux\_info {

uint32\_t simple\_sum;

uint32\_t level\_sum;

uint32\_t marker;

} aux\_info;

static void print\_node(tree\_node \*node) {

printf("(num: %ld, parent: %ld, depth: %ld) ", node->num,

node->parent != NULL ? node->parent->num : 0, node->depth);

}

void print\_tree(tree\_node \*nodes, size\_t count) {

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

tree\_node \*node = &nodes[i];

print\_node(node);

}

printf("\n");

}

static int order\_tree(const void \*lhs, const void \*rhs) {

tree\_node \*a = \*((tree\_node \*\*)lhs);

tree\_node \*b = \*((tree\_node \*\*)rhs);

return a->depth - b->depth;

}

static void add\_depth(tree\_node \*node) {

if (node->parent == NULL) {

node->depth = 0;

} else if (node->depth == -1) {

add\_depth(node->parent);

node->depth = node->parent->depth + 1;

}

}

int main() {

long num\_nodes, num\_queries;

scanf("%ld %ld", &num\_nodes, &num\_queries);

tree\_node \*nodes = calloc(num\_nodes, sizeof(tree\_node));

tree\_node \*\*order = calloc(num\_nodes, sizeof(tree\_node \*));

aux\_info \*info = calloc(num\_nodes, sizeof(aux\_info));

for (long i = 0; i < num\_nodes; ++i) {

tree\_node \*node = &nodes[i];

node->num = i + 1;

node->depth = -1;

order[i] = &nodes[i];

}

for (long i = 0; i < num\_nodes - 1; i++) {

long a, b;

scanf("%ld %ld", &a, &b);

tree\_node \*node\_a = &nodes[a - 1];

tree\_node \*node\_b = &nodes[b - 1];

if (node\_b->parent == NULL) {

node\_b->parent = node\_a;

} else if (node\_a->parent == NULL) {

node\_a->parent = node\_b;

} else {

exit(1);

}

}

for (long i = 0; i < num\_nodes; ++i) {

add\_depth(&nodes[i]);

}

qsort(order, num\_nodes, sizeof(tree\_node \*), order\_tree);

for (long i = 0; i < num\_queries; ++i) {

unsigned long k;

scanf("%ld", &k);

for (long j = 0; j < k; j++) {

long node\_num;

scanf("%ld", &node\_num);

info[node\_num - 1].marker = 1;

}

uint64\_t total = 0;

for (long j = num\_nodes - 1; j >= 0; --j) {

tree\_node node = \*order[j];

uint64\_t node\_num = node.num;

uint64\_t node\_index = node\_num - 1;

aux\_info node\_info = info[node\_index];

if (node\_info.marker == 0 && node.depth == 0) {

continue;

}

uint64\_t node\_simple\_sum = node\_info.simple\_sum;

uint64\_t node\_level\_sum = node\_info.level\_sum;

if (node\_info.marker != 0) {

// Add all the combintations made with this node and its children

total = total + node\_level\_sum \* node\_num;

if (total > LIMIT) {

total = total % LIMIT;

}

node\_simple\_sum += node\_num;

} else if (node\_simple\_sum == 0) {

continue;

}

// Increment the level

node\_level\_sum += node\_simple\_sum;

tree\_node \*parent = node.parent;

if (parent != NULL) {

uint64\_t parent\_index = parent->num - 1;

aux\_info parent\_info = info[parent\_index];

uint64\_t parent\_simple\_sum = parent\_info.simple\_sum;

uint64\_t parent\_level\_sum = parent\_info.level\_sum;

// Add the combinations that this subtree makes with all sibling

// subtrees processed so far

total = (total + (parent\_simple\_sum \* node\_level\_sum) +

(parent\_level\_sum \* node\_simple\_sum));

if (total > LIMIT) {

total = total % LIMIT;

}

parent\_simple\_sum = parent\_simple\_sum + node\_simple\_sum;

if (parent\_simple\_sum > LIMIT) {

parent\_simple\_sum = parent\_simple\_sum % LIMIT;

}

parent\_level\_sum = parent\_level\_sum + node\_level\_sum;

if (parent\_level\_sum > LIMIT) {

parent\_level\_sum = parent\_level\_sum % LIMIT;

}

info[parent\_index].simple\_sum = parent\_simple\_sum;

info[parent\_index].level\_sum = parent\_level\_sum;

}

}

memset(info, 0, sizeof(aux\_info) \* num\_nodes);

long ans = total;

printf("%ld\n", ans);

}

return 0;

}

Array and simple queries

#include <stdio.h>

#include <stdlib.h>

typedef struct \_ct\_node{

int size;

int priority;

int value;

struct \_ct\_node \*left,\*right;

} ct\_node;

void tar(ct\_node \*root);

int get\_size(ct\_node \*root);

ct\_node\* merge(ct\_node \*L,ct\_node \*R);

int sizeOf(ct\_node \*root);

void recalc(ct\_node \*root);

void split(int x,ct\_node \*\*L,ct\_node \*\*R,ct\_node \*root);

void computeTree();

int P[100000],T[100000],st[100000],N;

ct\_node pool[100000];

int main(){

int M,x,y,z,i;

ct\_node \*root,\*l,\*m,\*r,\*t;

scanf("%d%d",&N,&M);

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

scanf("%d",&pool[i].value);

P[i]=pool[i].priority=rand();

pool[i].left=pool[i].right=NULL;

}

computeTree();

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

if(T[i]==-1)

root=&pool[i];

else

if(i<T[i])

pool[T[i]].left=&pool[i];

else

pool[T[i]].right=&pool[i];

get\_size(root);

for(i=0;i<M;i++){

scanf("%d%d%d",&x,&y,&z);

switch(x){

case 1:

split(y-2,&l,&t,root);

split(z-y,&m,&r,t);

root=merge(merge(m,l),r);

break;

default:

split(y-2,&l,&t,root);

split(z-y,&m,&r,t);

root=merge(merge(l,r),m);

}

}

N=0;

tar(root);

printf("%d\n",(T[0]>T[N-1])?T[0]-T[N-1]:T[N-1]-T[0]);

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

printf("%d ",T[i]);

return 0;

}

void tar(ct\_node \*root){

if(!root)

return;

tar(root->left);

T[N++]=root->value;

tar(root->right);

return;

}

int get\_size(ct\_node \*root){

if(!root)

return 0;

root->size=get\_size(root->left)+get\_size(root->right)+1;

return root->size;

}

ct\_node\* merge(ct\_node \*L,ct\_node \*R){

if(!L)

return R;

if(!R)

return L;

if(L->priority>R->priority){

L->right=merge(L->right,R);

recalc(L);

return L;

}

R->left=merge(L,R->left);

recalc(R);

return R;

}

int sizeOf(ct\_node \*root){

return (root)?root->size:0;

}

void recalc(ct\_node \*root){

root->size=sizeOf(root->left)+sizeOf(root->right)+1;

return;

}

void split(int x,ct\_node \*\*L,ct\_node \*\*R,ct\_node \*root){

if(!root){

\*L=\*R=NULL;

return;

}

int curIndex=sizeOf(root->left);

ct\_node \*t;

if(curIndex<=x){

split(x-curIndex-1,&t,R,root->right);

root->right=t;

recalc(root);

\*L=root;

}

else{

split(x,L,&t,root->left);

root->left=t;

recalc(root);

\*R=root;

}

return;

}

void computeTree(){

int i,k,top=-1;

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

k=top;

while(k>=0 && P[st[k]]<P[i])

k--;

if(k!=-1)

T[i]=st[k];

if(k<top)

T[st[k+1]]=i;

st[++k]=i;

top=k;

}

T[st[0]]=-1;

return;

}

Median Updates

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#include <assert.h>

/\* Head ends here \*/

void insertionSort(int N,int \*x){

int temp = x[N-1];

int j=N-2;

while ( j>=0 && x[j]>temp){

x[j+1] = x[j];

j--;

}

x[j+1] = temp;

}

void median(int N,char (\*s), int \*x) {

int size = 0;

int a[N];

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

if (s[i] == 'a'){

a[size++] = x[i];

insertionSort(size,&a[0]);

if (size % 2 == 1)

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

else{

if ((a[size/2]+a[(size/2)-1]) % 2 == 0)

printf("%ld",((long)a[size/2]+a[(size/2)-1])/2);

else

printf("%1.1f",((signed long)a[size/2]+a[(size/2)-1])/2.0);

}

printf("\n");

}

else{

if (size == 0) {printf("Wrong!\n");continue;}

int found = 0;

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

if (a[j] == x[i]){

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

a[k] = a[k+1];

}

size--;

if (size == 0) printf("Wrong!");

else if (size % 2 == 1)

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

else{

if ((a[size/2]+a[(size/2)-1]) % 2 == 0)

printf("%ld",((long)a[size/2]+a[(size/2)-1])/2);

else

printf("%1.1f",((signed long)a[size/2]+a[(size/2)-1])/2.0);

}

printf("\n");

found = 1;

break;

}

}

if (!found) printf("Wrong!\n");

}

}

}

int main(){

//Helpers for input/output

int i, N;

scanf("%d", &N);

char s[N];

int x[N];

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

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

}

median(N,s,x);

}

Balanced Forest

include <assert.h>

#include <limits.h>

#include <math.h>

#include <stdbool.h>

#include <stddef.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* readline();

char\*\* split\_string(char\*);

struct Node

{

int64\_t sum; // sum of all node in this tree.

int64\_t testSum;

int data;

int parent;

int \*child;

int childCnt;

};

typedef struct Node Node;

void sumTree(Node \*root, int index)

{

int i;

root[index].sum = root[index].data;

for (i = 0; i < root[index].childCnt; i++)

{

int child = root[index].child[i];

if (child == root[index].parent) continue;

sumTree(root, child);

root[index].sum += root[child].sum;

}

root[index].testSum = root[index].sum;

}

void updateTree(Node \*tree, int root, int parent)

{

tree[root].parent = parent;

int i;

for (i = 0; i < tree[root].childCnt; i++)

{

if (tree[root].child[i] == parent) continue;

updateTree(tree, tree[root].child[i], root);

}

}

int64\_t childSum(Node \*tree, int root, int branch\_root, int64\_t targetSum, bool \*bFound)

{

int i;

int64\_t currSum = 0;

if (tree[root].testSum < targetSum) return tree[root].testSum;

for (i = 0; (i < tree[root].childCnt) && (\*bFound==0); i++)

{

int child = tree[root].child[i];

if (child == tree[root].parent) continue;

if (child == branch\_root) continue;

int64\_t chSum = childSum(tree, child, branch\_root, targetSum, bFound);

if (chSum == targetSum)

{

\*bFound = 1;

break;

}

currSum += chSum;

}

return currSum + tree[root].data;

}

// Complete the balancedForest function below.

int64\_t balancedForest(int c\_count, int\* c, int edges\_rows, int edges\_columns, int\*\* edges) {

int i, j;

// build tree.

Node \*tree = (Node \*)calloc(c\_count, sizeof(Node));

for (i = 0; i < c\_count; i++)

{

tree[i].data = c[i];

tree[i].childCnt = 0;

tree[i].child = NULL;

tree[i].parent = -1;

tree[i].sum = 0;

}

for (i = 0; i < edges\_rows; i++)

{

int pa = edges[i][0] - 1;

int ch = edges[i][1] - 1;

tree[pa].child = (int \*)realloc(tree[pa].child, (1 + tree[pa].childCnt) \* sizeof(int));

tree[pa].child[tree[pa].childCnt] = ch;

tree[pa].childCnt++;

tree[ch].child = (int \*)realloc(tree[ch].child, (1 + tree[ch].childCnt) \* sizeof(int));

tree[ch].child[tree[ch].childCnt] = pa;

tree[ch].childCnt++;

}

// Now update the parent\_node;

int root = 0; // pick the first one as root.

updateTree(tree, root, -1);

sumTree(tree, root);

int64\_t treeSum = 0;

treeSum = tree[root].sum;

int64\_t maxSum = (treeSum - 1) / 2 + 1;

int64\_t minSum = treeSum / 3 - 1;

int64\_t minW = -1;

for (i = 0; i < c\_count; i++)

{

if (i == root) continue;

//if (tree[i].sum >= minSum && tree[i].sum <= maxSum)

{

int64\_t sumI = tree[i].sum;

// Check for special case.

int64\_t sumJ = treeSum - sumI;

if (sumI == sumJ)

{

if (minW<0 || minW>sumI) minW = sumI;

}

else

{

bool bFound = 0;

int64\_t targetSum;

int searchRoot = root;

int branchRoot = i;

int64\_t w = 0;

if (sumI > sumJ)

{

targetSum = sumI;

sumI = sumJ;

sumJ = targetSum;

searchRoot = i;

branchRoot = root;

}

if ((sumI << 1) < sumJ)

{

targetSum = sumJ >> 1;

if (sumJ - targetSum != targetSum) continue;

w = targetSum - sumI;

}

else

{

targetSum = sumI;

w = targetSum - (sumJ - sumI);

}

if (minW >= 0 && minW < w) continue;

// search in the main tree

// first, update the testSum;

if (searchRoot == root)

{

int curr = tree[branchRoot].parent;

int64\_t branchSum = tree[branchRoot].sum;

while (curr != -1)

{

tree[curr].testSum -= branchSum;

curr = tree[curr].parent;

}

}

childSum(tree, searchRoot, branchRoot, targetSum, &bFound);

if (bFound)

{

if (minW == -1 || minW > w) minW = w;

}

// last, restore the testSum

if (searchRoot == root)

{

int curr = tree[branchRoot].parent;

while (curr != -1)

{

tree[curr].testSum = tree[curr].sum;

curr = tree[curr].parent;

}

}

}

}

}

return minW;

}

int main()

{

FILE\* fptr = fopen(getenv("OUTPUT\_PATH"), "w");

char\* q\_endptr;

char\* q\_str = readline();

int q = strtol(q\_str, &q\_endptr, 10);

if (q\_endptr == q\_str || \*q\_endptr != '\0') { exit(EXIT\_FAILURE); }

for (int q\_itr = 0; q\_itr < q; q\_itr++) {

char\* n\_endptr;

char\* n\_str = readline();

int n = strtol(n\_str, &n\_endptr, 10);

if (n\_endptr == n\_str || \*n\_endptr != '\0') { exit(EXIT\_FAILURE); }

char\*\* c\_temp = split\_string(readline());

int\* c = malloc(n \* sizeof(int));

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

char\* c\_item\_endptr;

char\* c\_item\_str = \*(c\_temp + i);

int c\_item = strtol(c\_item\_str, &c\_item\_endptr, 10);

if (c\_item\_endptr == c\_item\_str || \*c\_item\_endptr != '\0') { exit(EXIT\_FAILURE); }

\*(c + i) = c\_item;

}

int c\_count = n;

int\*\* edges = malloc((n - 1) \* sizeof(int\*));

for (int i = 0; i < n - 1; i++) {

\*(edges + i) = malloc(2 \* (sizeof(int)));

char\*\* edges\_item\_temp = split\_string(readline());

for (int j = 0; j < 2; j++) {

char\* edges\_item\_endptr;

char\* edges\_item\_str = \*(edges\_item\_temp + j);

int edges\_item = strtol(edges\_item\_str, &edges\_item\_endptr, 10);

if (edges\_item\_endptr == edges\_item\_str || \*edges\_item\_endptr != '\0') { exit(EXIT\_FAILURE); }

\*(\*(edges + i) + j) = edges\_item;

}

}

int edges\_rows = n - 1;

int edges\_columns = 2;

int64\_t result = balancedForest(c\_count, c, edges\_rows, edges\_columns, edges);

fprintf(fptr, "%lld\n", result);

}

fclose(fptr);

return 0;

}

char\* readline() {

size\_t alloc\_length = 1024;

size\_t data\_length = 0;

char\* data = malloc(alloc\_length);

while (true) {

char\* cursor = data + data\_length;

char\* line = fgets(cursor, alloc\_length - data\_length, stdin);

if (!line) {

break;

}

data\_length += strlen(cursor);

if (data\_length < alloc\_length - 1 || data[data\_length - 1] == '\n') {

break;

}

alloc\_length <<= 1;

data = realloc(data, alloc\_length);

if (!line) {

break;

}

}

if (data[data\_length - 1] == '\n') {

data[data\_length - 1] = '\0';

data = realloc(data, data\_length);

} else {

data = realloc(data, data\_length + 1);

data[data\_length] = '\0';

}

return data;

}

char\*\* split\_string(char\* str) {

char\*\* splits = NULL;

char\* token = strtok(str, " ");

int spaces = 0;

while (token) {

splits = realloc(splits, sizeof(char\*) \* ++spaces);

if (!splits) {

return splits;

}

splits[spaces - 1] = token;

token = strtok(NULL, " ");

}

return splits;

}