Shu Lin <linshu123@gmail.com>

August 19, 2013 2:20 PM

(No Subject)

```
//
//  main.cpp
//  test
//
//  Created by Shu Lin on 9/28/12.
//  Copyright (c) 2012 Shu Lin. All rights reserved.
//

#include <iostream>
#include <list>
#include <vector>
#include <stack>
using namespace std;

#define maxn 1000

struct Node{
        Node *next;
        int value;
    Node(int n):value(n), next(nullptr){}
};

struct List {
    Node *first;
};

void remove_duplicate(List a){
        Node *node_to_check = a.first;
        Node *cur_node, *pre_node;

        while (node_to_check){
                cur_node = node_to_check->next;
                pre_node = node_to_check;
                while (cur_node){
                        if (cur_node->value == node_to_check->value){

                                //remove cur_node
                                Node *tmp = cur_node;
                                pre_node->next = cur_node->next;
                                delete tmp;

                        }
                        pre_node = cur_node;
                        cur_node = cur_node->next;
                }
                node_to_check = node_to_check->next;
        }
        return;
}

bool all_unique(vector<int> *Array){
    int original;
    bool unique = 1;

    for (int i  = 0; i < Array->size(); i++) {
        original = (*Array)[i];
        for (int j = i + 1; j < Array->size(); j++) {
            if (original == (*Array)[j]) {
                unique = 0;
            }
        }
    }
    return unique;
}

string reverse_str(string str){

    int n = int(str.size() - 1);
```

```
    for (int i = 0; i <= n/2; i++) {
        char tmp = str[i];
        str[i] = str[n - i];
        str[n - i] = tmp;
    }

    return str;
}
string remove_duplicate(string str){
    char cur_char;

    for (int i = 0; i < str.size(); i++) {
        cur_char = str[i];
        for (int j = i + 1; j < str.size(); j++) {
            if (str[j] == cur_char) {
                str.erase(j, 1);
                j--;
            }
        }
    }

    return str;
}

bool is_anagram(string a, string b){
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    return (a == b);
}

string replace(string str, string to_replace, string new_content){
    while (size_t pos = str.find(to_replace, 0)) {
        if (pos == -1)
            return str;
        str.erase(pos, to_replace.size());
        str.insert(pos, new_content);
    }
    return str;
}

void delete_node(Node *node_to_del){

    if(!node_to_del->next)
        exit(1);
    Node *tmp = node_to_del->next;
    node_to_del->value = tmp->value;
    node_to_del->next = tmp->next;
    delete tmp;
    return;
}

class SetOfStacks{
    vector<stack<int> > stackArray;
    int limit;
public:
    int pop();
    void push(int n);
    SetOfStacks(int n);
};

SetOfStacks::SetOfStacks(int n){
    stackArray.push_back(stack<int>());
    limit = n;
};

int SetOfStacks::pop(){
    if (stackArray.end()->empty()) {
        stackArray.erase(stackArray.end());
    }
    return stackArray.end()->top();
};

void SetOfStacks::push(int n){
    if (stackArray.end()->size() > limit){
        stack<int> s;
        stackArray.push_back(s);
    }
```

```
    stackArray.end()->push(n);
};

unsigned int ELFHash(string& str)
{
    unsigned int hash = 0;
    unsigned int x    = 0;

    for(size_t i = 0; i < str.length(); i++)
    {
        hash = (hash << 4) + str[i];
        if((x = hash & 0xF0000000L) != 0)
        {
            hash ^= (x >> 24);
        }
        hash &= ~x;
    }

    return hash;
}
/* End Of ELF Hash Function */

// KMP algorithm.
namespace KMP {
    vector<int> KMPnext(string str){
        vector<int> next(str.size());
        int i = 0, j = -1;
        next[i] = j;
        while (i < str.size()) {
            while (j >= 0 && str[i] != str[j]) {
                j = next[j];
            }
            i++;
            j++;
            next[i] = j;
        }
        return next;
    }

    bool KMP(string nstr, string pat, vector<int> & next){
        int i = 0, j = 0;
        while (i < nstr.size()) {
            while (nstr[i] != pat[j] && j >= 0) {
                j = next[j];
            }
            i++;
            j++;
            if (j == pat.size())
                return true;
        }
        return false;
    }

}

bool same(int a, int b){
    return a == b;
}

bool lessthan(int a, int b){
    return a < b;
}

void print_vec(vector<int> & array){

    for (size_t i = 0; i < array.size(); i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
    return;
}

// Counting sort a string.
string cnt_sort(string const &str){
    int cnt[256];
    string result;
    memset(cnt, 0, sizeof(cnt));
```

```
    for (size_t i = 0; i < str.size(); i++) {
        cnt[str[i]]++;
    }
    for (size_t i = 0; i < 256; i++) {
        if (!cnt[i]) continue;
        while (cnt[i]--) {
            result += i;
        }
    }
    return result;
}


void reverse_str_c(char k[]){
    size_t n = strlen(k) - 1;

    for (size_t i = 0; i <= (n >> 1); i++) {

        // if k[i] == k[n - i], ^= will set k[i] to null,
        // which will change the string length.

        if (k[i] != k[n - i]){
            k[i] ^= k[n - i];
            k[n - i] ^= k[i];
            k[i] ^= k[n - i];
        }
    }
    return;
}
stack<int> a;

int Q::dequeue(){
    stack<int> b;
    while (!a.empty()) {
        b.push(a.top());
        a.pop();
    }
    int ans = b.top();
    b.pop();
    while (!b.empty()) {
        a.push(b.top());
        b.pop();
    }
    return ans;
}

int factorial(int n){
    if (n == 1 || n == 0){
        return 1;
    }
    else
        return n * factorial(n - 1);
}

// Suffix array to for string problems.
namespace SuffixArray_ha {

    int wa[maxn], wb[maxn];
    char r[maxn] = "aabaaaab";
    int cnt[maxn], wv[maxn];
    int sa[maxn];

    bool cmp(int r[], int a, int b, int j){
        return r[a] == r[b] && r[a + j] == r[b + j];
    }

    void da(int n, int m){
        int i, j, p, *x = wa, *y = wb, *t;

        // Radix sort
        memset(cnt, 0, sizeof(cnt));
        for (i = 0; i < n; i++) cnt[x[i] = r[i]]++;
        for (i = 1; i < m; i++) cnt[i] += cnt[i - 1];
        for (i = n - 1; i>=0; i--) sa[--cnt[x[i]]]= i;

        //
        for (p = 1, j = 1; p<n; j<<=1, m=p) {
```

```
            // get the rank from the previous radix sort.
            for (p = 0, i = n - j; i < n; i++) y[p++] = i;
            for (i = 0; i < n; i++) if (sa[i] >= j) y[p++] = sa[i] - j;
            for (i = 0; i < n; i++) wv[i] = x[y[i]];
            memset(cnt, 0, sizeof(cnt));
            for (i = 0; i < n; i++) cnt[wv[i]]++;
            for (i = 1; i < m; i++) cnt[i] += cnt[i - 1];
            for (i = n - 1; i>=0; i--) sa[--cnt[wv[i]]]= y[i];
            for (i = 1, t=x, x=y, y=t, p = 1, x[sa[0]] = 0; i < n; i++) {
                x[sa[i]] = cmp(y, sa[i], sa[i-1], j)?p-1:p++;
            }
        }
        return;
    }

    int rank[maxn], height[maxn];

    void calheight(int n){
        int i, j, k = 0, tmp;

        // Calculate rank[].
        for (i = 0; i < n; rank[sa[i]] = i, i++);

        // See how much two suffixes overlap.
        for (i = 0; i < n; i++){
            k?k--:0, j = sa[rank[i] - 1];
            while(r[j + k] == r[i + k])
                k++;

            height[rank[i]] = k;
        }
        return;
    }
}


// Counting sort.
string cnt_sort(string const &str){
    int cnt[256];
    string result;
    memset(cnt, 0, sizeof(cnt));
    for (size_t i = 0; i < str.size(); i++) {
        cnt[str[i]]++;
    }
    for (size_t i = 0; i < 256; i++) {
        if (!cnt[i]) continue;
        while (cnt[i]--) {
            result += i;
        }
    }
    return result;
}

// This function finds the longest contiguous part
// of an array that adds up to a maximum sum. Empty
// contiguous part (sum to 0) is allowed.

int find_max(int array[], int n){
    int cur_max = 0, cur_sum = 0;
    for (int i = 0; i < n; i++) {
        cur_sum += array[i];
        cur_max = max(cur_max, cur_sum);
        cur_sum = max(cur_sum, 0);
    }
    return cur_max;
}

// partition a linked list, so that first part are all numbers less than x,
// second part all numbers larger or equal to x.
void partition(int x, list<int> &myList)
{
    auto p1 = myList.begin(), p2 = myList.begin();
    int p1_step = 0, p2_step = 0;

    while (p1 != myList.end() && p2 != myList.end()) {

        while (p1 != myList.end() && *p1 < x) {
```

```
            ++p1;
            ++p1_step;
        }
        while (p2 != myList.end() && *p2 >= x) {
            ++p2;
            ++p2_step;
        }

        if (p1 != myList.end() && p2 != myList.end()){
            if (p1_step < p2_step){
                swap(*p1, *p2);
                //                  printList(myList);
            }
            else{
                cout << "swap pointer" << endl;
                swap(p1, p2);
                swap(p1_step, p2_step);
            }
        }
    }
}

// RMQ algorithm.
namespace RMQ {

    // How to select between two numbers.
    int choose(int a, int b){
        return max(a, b);
    }

    // Preprocess.
    void makeTable(const vector<int> &array, vector<vector<int>> &table){

        // Locate space;
        int logSize = int(log(array.size()) / log(2.0));
        for (int i = 0; i < array.size(); ++i) {
            table.push_back(vector<int>(logSize, 0));
        }

        // Initialize.
        for (int i = 0; i <array.size(); ++i) {
            table[i][0] = array[i];
        }

        for (int j = 1; (1 << j) <= array.size(); ++j) {
            for (int i = 0; i + (1 << j) - 1 < array.size(); ++i) {
                table[i][j] = choose(table[i][j - 1], table[i + (1<<(j-1))][j - 1]);
            }
        }
    }

    // Return the max/min element in the range [a, b]
    int RMQ(int a, int b, const vector<vector<int>> &table){
        int k = int(log(double(b - a + 1))/log(2.0));
        return choose(table[a][k], table[b - (1<<k) + 1][k]);
    }
}

// Manacher's Algorithm to find longest palindrome

namespace Manacher {

    const char special_sign_c = 1;
    char r[maxn]; // Original string.
    char s[2 * maxn]; // New string after special character inserted.
    int p[2 * maxn];  // Record the half length of the longest
                      // palindrome centered at s[i].

    int manacher(char r[]){

        int n = (int)strlen(r), i, j, maxLen, mx, id;

        // Preprocess, add a special sign at the beginning and after every char.
        for (j = 0, i = 0; i < n; i++) {
            s[j++] = special_sign_c;
            s[j++] = r[i];
        }
```

```
        s[j++] = special_sign_c;

        // Manacher's algorithm.
        memset(p, 0, sizeof(p));
        mx = 0, id = 0;
        for (i = 1; i < j; i++) {   // j is the length of the string in s.
            p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1; // 2 * id - i is i关于id的对称点.
            while ((s[i - p[i]] == s[i + p[i]]) && i >= p[i]) p[i]++;
            if (mx < i + p[i]){
                mx = i + p[i];  // mx is the farthest point we ever looked.
                id = i;         // id is the central of the palindrome that touches mx.
            }
        }

        // Return max_value - 1.
        maxLen = *max_element(p + 1, p + j);
        return maxLen - 1;
    }

    // If wanna return the longest substring itself,
    // we can get the largest p[i], set maxInd = i, maxLength = p[i]
    // then map values of i, p[i] back to original string:
    // take substr(maxInd/2 - (maxLength - 1)/2, maxLength - 1);

}

// SuffixArray implementation from Stanford ACM notebook.
// Complexity: O(nlog(n)^2)
namespace SuffixArray_HA_Implementation2 {
    const int maxn = 65536;
    const int maxlg = 17;

    struct Entry {
        int nr[2], p;
    }compArray[maxn];

    int cmp(const Entry& a, const Entry& b){
        if (a.nr[0] == b.nr[0])
            return a.nr[1] < b.nr[1];
        return a.nr[0] < b.nr[0];
    }

    int daArray[maxlg][maxn];
    int sa[maxn];

    void getSA(char str[]){
        int n = (int)strlen(str);

        // First populate the first layer of daArray. (Since we don't need any
        // comparison right now.
        for (int i = 0; i < n; ++i) {
            daArray[0][i] = str[i] - 'a';
        }

        // Now start to sort the array. In each iteration, we need to populate
        // compArray and do the sorting. Where do we get the data to populate?
        // from last level of daArray. Also, stp starts at 1, and increment each
        // iteration, but cnt starts at 1 and increment to twice as big each
        // iteration. The loop stops when cnt is exceeding the size of the array.
        int stp = 1, cnt = 1;
        for (; cnt <= n ; stp++, cnt <<= 1) {
            for (int i = 0; i < n; ++i) {
                compArray[i].nr[0] = daArray[stp - 1][i];
                compArray[i].nr[1] = i+cnt<n ? daArray[stp - 1][i + cnt] : -1;
                compArray[i].p = i;
            }
            sort(compArray, compArray + n, cmp);

            // After sorting, we have to put the information back to daArray, at
            // current level.
            for (int i = 0; i < n; ++i)
            {
                if (i > 0 && compArray[i].nr[0] == compArray[i-1].nr[0] &&
                    compArray[i].nr[1] == compArray[i-1].nr[1])
                {
                    daArray[stp][compArray[i].p] = daArray[stp][compArray[i - 1].p];
                }
```

```cpp
                else
                    daArray[stp][compArray[i].p] = i;
            }
        }
        stp--;
        for (int i = 0; i < n; ++i) {
            sa[daArray[stp][i]] = i;
        }
    }

}

namespace BinaryTreeRelated {
    struct TreeNode {
        int val;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int n): val(n), left(nullptr), right(nullptr){}
    };

    // Get height of a tree.
    int height(TreeNode *tree, int curHeight){
        if (!tree)
            return 0;
        return max(height(tree->left, curHeight + 1),
                   height(tree->right, curHeight + 1)) + 1;
    }

    // Return if a tree is balanced.
    bool isBalanced(TreeNode *tree){
        if (!tree)
            return true;
        if (abs(height(tree->left, 0) - height(tree->right, 0)) > 1)
            return false;
        return isBalanced(tree->left) && isBalanced(tree->right);
    }

    // Create a tree
    void creatTreeHelper(TreeNode *tree, int n)
    {
        static int curNum = 1;
        if (curNum >= n)
            return;
        tree->left = new TreeNode(0);
        curNum++;
        if (curNum >= n)
            return;
        tree->right = new TreeNode(0);
        curNum++;

        creatTreeHelper(tree->left, n);
        creatTreeHelper(tree->right, n);
    }

    // Create a binary tree with n nodes and minimum height
    TreeNode *createTree(int n){
        TreeNode *tree = new TreeNode(0);
        creatTreeHelper(tree, n);
        return tree;
    }

    // Fill a binary tree with nodes from an array. The tree has to be same size
    // as the array, and is balanced.
    void fillTree(int array[], int n, TreeNode *tree){
        static int cur = 0;
        if (!tree)
            return;
        fillTree(array, n, tree->left);
        tree->val = array[cur++];
        if (cur == n)
            return;
        fillTree(array, n, tree->right);
    }

    // Inorder traverse and print node of a binary tree.
    void printTree(TreeNode *tree){
        if (!tree)
```

```
            return;
        printTree(tree->left);
        cout << tree->val << " ";
        printTree(tree->right);
    }

    // Print the nodes that a DFS visited, including back-track visit.
    void printDFSRoute(TreeNode *tree, vector<int> &DFSRoute){
        if (!tree)
            return;
        DFSRoute.push_back(tree->val);

        if (tree->left){
            printDFSRoute(tree->left, DFSRoute);
            DFSRoute.push_back(tree->val);
        }

        if (tree->right){
            printDFSRoute(tree->right, DFSRoute);
            DFSRoute.push_back(tree->val);
        }
        return;
    }

    bool isSameTree(TreeNode *p, TreeNode *q) {
        // Start typing your C/C++ solution below
        // DO NOT write int main() function

        if (!p && !q)
            return true;

        if (p && !q)
            return false;

        if (!p && q)
            return false;

        if (p->val == q->val)
            return isSameTree(p->left, q->left) &&
                    isSameTree(p->right, q->right);

        else
            return false;

    }

    TreeNode* buildSameTree(TreeNode *tree){
        if (!tree)
            return nullptr;
        TreeNode * node = new TreeNode(tree->val);
        node->left = buildSameTree(tree->left);
        node->right = buildSameTree(tree->right);
        return node;
    }

}

//Symmetric Tree
namespace IsSymmetric {
    struct TreeNode {
        int val;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int x) : val(x), left(NULL), right(NULL) {}
    };

    bool isPalindrome(const vector<int> &trav){
        int i = 0;
        int k = (int)trav.size() - 1;
        while (i < k) {
            if (trav[i] != trav[k])
                return false;
            i++;
            k--;
        }
        return true;
    }
```

```
    class Solution {
    public:

        vector<int> trav;

        void traversal(TreeNode *root){
            if (!root)
                return;
            traversal(root->left);
            trav.push_back(root->val);
            traversal(root->right);
        }

        bool isSymmetric(TreeNode *root) {
            // Start typing your C/C++ solution below
            // DO NOT write int main() function
            if (!root){
                return true;
            }

            traversal(root);
            return isPalindrome(trav);
        }
    };
}


// Get the max sum that could be taken from some range of a vector.
int getMaxSumRange(const vector<int> &vec){
    queue<int> myQueue;
    int queueSum = 0;
    int maxSum = 1 << (sizeof(int) * 8 - 1); // Get minimum int.

    // Keep pushing element on the queue, and add up. If
    // there's one point the sum is smaller than zero, pop until it's larger
    // than zero.
    for (int i = 0; i < vec.size(); ++i)
    {
        myQueue.push(vec[i]);
        queueSum += vec[i];
        if (queueSum > maxSum)
            maxSum = queueSum;

        while (queueSum < 0){
            queueSum -= myQueue.front();
            myQueue.pop();
        }
    }

    return maxSum;
}

// The idea behind this algorithm is that every time we put in a new element
// we append it to the end of all previous subsets(including empty set), and
// these new subsets together with those previous subsets will be all subsets
// when the new element is added.
namespace GetAllSubsets {

    // Add one element to the back of all lists.
    vector<vector<int>> allEle2AllSet(const vector<vector<int>>& original, int ele){
        auto ans = original;
        ans.push_back(vector<int>()); // Add an empty set.
        for_each(ans.begin(), ans.end(), [ele](vector<int> &oneVec){
            oneVec.push_back(ele);
        });
        return ans;
    }

    // Return all subsets of a set.
    vector<vector<int>> getAllSubset(const set<int> &set){
        vector<int> setVec(set.begin(), set.end());
        vector<vector<int>> ans;
        for (int i = 0; i < set.size(); i++) {
            auto newSubsets = allEle2AllSet(ans, setVec[i]);
            ans.insert(ans.end(), newSubsets.begin(), newSubsets.end());
```

```
        }
        return ans;
    }

    void printAllSubset(const vector<vector<int>> &subsets){
        cout << endl; // Indicate empty set.
        for_each(subsets.begin(), subsets.end(), [](const vector<int> &subset){
            for_each(subset.begin(), subset.end(), [](int n){
                cout << n << " ";
            });
            cout << endl;
        });
    }
}

// The idea is insert new element one by one into all locations of previous
// permutations.
namespace GetAllPermutations {

    // Insert the element at all possible locations inside vec. return
    // result in an array of vectors.
    vector<vector<int>> insertEleAtAllLocations(const vector<int>& vec, int ele){
        vector<vector<int>> ans;
        for (int i = 0; i < vec.size(); ++i) {
            auto newVec = vec;
            newVec.insert(newVec.begin() + i, ele);
            ans.push_back(newVec);
        }
        ans.push_back(vec);
        (ans.end() - 1)->push_back(ele);
        return ans;
    }

    // Return all permutations of a vector.
    vector<vector<int>> permutations(const vector<int> vec){
        vector<vector<int>> ans;
        ans.push_back(vector<int>());
        // For every single element in vector, insert it to previous permutations.
        for (int i = 0; i < vec.size(); ++i) {
            vector<vector<int>> newAns;
            for (int j = 0; j < ans.size(); ++j) {
                auto newVec = insertEleAtAllLocations(ans[j], vec[i]);
                newAns.insert(newAns.end(), newVec.begin(), newVec.end());
            }
            ans = newAns;
        }
        return ans;
    }

    // Print all permutations of a vector.
    void printAllPermutations(const vector<int>& vec){
        vector<vector<int>> ans = permutations(vec);
        for_each(ans.begin(), ans.end(), [](const vector<int> &onePerm){
            for_each(onePerm.begin(), onePerm.end(), [](int n){
                cout << n << " ";
            });
            cout << endl;
        });
    }
}

//有一种很基本的动态规划是给你某一套面值的钱币, 比如1, 10, 15, 20, 21, 25,
//然后给你某个总额, 比如63, 让你求最少用多少张钱币可以组成这个面值.
namespace minCoinsNeeded {

    // 这种题, 设一个dp[]数组, dp[i]表示组成总面额为i需要的最少钱币数量. 那么做这个循环:

    // n为我们要求的总面额, faceVal[]表示所有不同的面额, 总共有m种.
    // Initialize everthing to be maximum value possible (choose face value 1 only).

    int dp[1000]; // dp array.
    int n = 63; // total value.
    int m = 6;  // total number of denomination
    int faceVal[] = {1, 10, 15, 20, 21, 25}; // different denominations.
    for (int i = 0; i <= n; i++)
        dp[i] = i;
```

```
    for (int i = 1; i <= n; i++){
        for (int j = 0; j < m; j++){
            if (i >= faceVal[j])
                dp[i] = min(dp[i - faceVal[j]] + 1, dp[i]);
        }
    }
}

//给你面值为1, 10, 25cents的硬币, 求有多少种不同的方式来表示n cents.
namespace totalNumWaysToPresentValue {
    int dp[10000];
    int n = 100;
    vector<int> faceVal = {1, 10, 25};
    int m = (int)faceVal.size();

    for (int i = 0; i <= n; i++)
        dp[i] = 1;


    for (int j = 1; j < m; j++){
        for (int i = 0; i <= n; i++){
            if (i >= faceVal[j])
                dp[i] += dp[i - faceVal[j]];
        }
    }
}

// Print all possible legal patterns of a set of parens. For example, if given
// 3 pairs of parens, all legal patterns are ((())) (()()) (())() ()(()) ()()()
// Idea: As long as there's left paren left, we can add it to the end of string,
// while the whole string remains legal. As long as the number of right paren
// left is bigger or equal to the number of left paren left, we can add it to
// the end of string while whole string remains legal.
namespace printAllParenPatterns {
    int total;
    void addParen(string str, char paren, int leftParen, int rightParen)
    {
        str += paren;
        paren == '(' ? leftParen-- : rightParen--;

        if (leftParen > 0){
            addParen(str, '(', leftParen, rightParen);
        }

        if (rightParen > 0 && rightParen > leftParen){
            addParen(str, ')', leftParen, rightParen);
        }

        if (!rightParen){
            cout << str << " ";
            total++;
        }
    }

    int printAllPossibleParen(int n){
        total = 0;
        addParen("", '(', n, n);
        cout << endl;
        return total;
    }
}


// Quick sort using iteration.
void quick_sort(int start, int end){

    struct Partition {
        int l;
        int r;
        Partition(){l = 0, r = 0;}
        Partition(int start, int end){l = start, r = end;}
    };

    stack<Partition> mystack;
    int i, j;
    mystack.push(Partition(start, end));
```

```
    while (!mystack.empty()) {

        Partition cur_part = mystack.top();
        i = cur_part.l;
        j = cur_part.r;
        mystack.pop();

        if (i >= j)
            continue;
        j++;

        while (i < j) {
            while (++i <= cur_part.r && myArray[i] <= myArray[cur_part.l]);
            while (--j > cur_part.l && myArray[j] >= myArray[cur_part.l]);
            if (i < j)
                swap(myArray[i], myArray[j]);
        }
        swap(myArray[cur_part.l], myArray[j]);
        mystack.push(Partition(cur_part.l, j));
        mystack.push(Partition(i, cur_part.r));
    }
}

// Given an array with a middle part unsorted, find the minimum range that has
// to be sorted in order to have the entire array sorted.
namespace MinRangeToBeSort {
    bool increasing(int a , int b){
        return a <= b;
    }

    bool decreasing(int a, int b){
        return a >= b;
    }

    // Find the part from the beginning of array that doesn't need to be touched
    // if the entire array is to be sorted in the order provided by cmp.
    int findBeginning(const vector<int> &array, bool(*cmp)(int, int)){
        stack<int> myStack;
        myStack.push(array[0]);
        int i = 1;
        while (cmp(myStack.top(), array[i])) {
            myStack.push(array[i]);
            i++;
        }

        int beginning = (int)myStack.size();
        for (; i < array.size(); ++i)
        {
            // If stack is empty, push the element onto it.
            if (myStack.empty())
                myStack.push(array[i]);

            // If the element does not defy increasing
            // order of the stack, push it on top.
            else if (cmp(myStack.top(), array[i]))
                myStack.push(array[i]);

            // If the element defy the increasing order, pop until we get to the point
            // where order requirement is satisfied.
            else {
                while (!cmp(myStack.top(), array[i]) && !myStack.empty()) {
                    myStack.pop();
                }
                myStack.push(array[i]);
                if (myStack.size() < beginning)
                    beginning = (int)myStack.size();
            }
        }
        return beginning - 1;
    }

    // Given an array with a middle part unsorted, find the minimum range that has
    // to be sorted in order to have the entire array sorted.
    void printRangeToBeSorted(const vector<int>& array){
        vector<int> arrayRe(array.rbegin(), array.rend());
        cout << findBeginning(array, increasing) << endl;
```

```
        cout << array.size() - 1 - findBeginning(arrayRe, decreasing) << endl;
    }
}

// Inplace algorithm to convert a binary tree into double linked list
// preserving inorder-traversal order.
namespace ConvertBinaryTreeToDoubleLinkedList {
    struct BiNode {
        BiNode(int val_): val(val_), node1(nullptr), node2(nullptr){}
        int val;
        BiNode *node1, *node2;

    };

    struct MaxMinNode {
        BiNode *min, *max;
        MaxMinNode(): min(nullptr), max(nullptr){}
    };

    void concateNode(BiNode *a, BiNode *b){
        a->node2 = b;
        b->node1 = a;
    }

    MaxMinNode toList(BiNode *root){
        if (!root){
            return MaxMinNode();
        }
        auto leftPart = toList(root->node1);
        auto rightPart = toList(root->node2);
        if (leftPart.max)
            concateNode(leftPart.max, root);
        if (rightPart.min)
            concateNode(root, rightPart.min);

        MaxMinNode ret;
        ret.max = rightPart.max ? rightPart.max : root;
        ret.min = leftPart.min ? leftPart.min : root;
        return ret;
    }
}

// Map a vector such as {1, 3000, 200, 600000, 5}
// to a vector of smaller integers such as {0, 3, 2, 4, 1}
namespace Discretization {

    struct Entry {
        int val;
        int originalIndex;
        Entry(int val_, int ind_): val(val_), originalIndex(ind_){};
    };

    bool cmp(const Entry&a, const Entry&b){
        return a.val < b.val;
    }

    vector<int> Discretize( const vector<int>& input2){

        vector<Entry> input;
        for (int i = 0; i < input2.size(); ++i) {
            input.push_back(Entry(input2[i], i));
        }

        sort(input.begin(), input.end(), cmp); // cmp compares input[i].val
        vector<int> newElements(input.size());

        for(int i = 0; i < input.size(); i++)
            newElements[input[i].originalIndex] = i;
    }
}

// Given the root of a binary tree, build a new tree with the exact same
// structures and values.
TreeNode* buildSameTree(TreeNode *tree)
{
    if (!tree)
        return nullptr;
```

```
    TreeNode * node = new TreeNode(tree->val);

    node->left = buildSameTree(tree->left);
    node->right = buildSameTree(tree->right);

    return node;
}

// See if a tree is symmetric.
namespace IsSymmetric {
    bool isPalindrome(const vector<int> &trav){
        int i = 0;
        int k = (int)trav.size() - 1;
        while (i < k) {
            if (trav[i] != trav[k])
                return false;
            i++;
            k--;
        }
        return true;
    }


    class Solution {
    public:

        vector<int> trav;

        // Inorder traversal
        void traversal(TreeNode *root){
            if (!root)
                return;
            traversal(root->left);
            trav.push_back(root->val);
            traversal(root->right);
        }

        // Return if a tree is symmetric.
        bool isSymmetric(TreeNode *root) {
            // Start typing your C/C++ solution below
            // DO NOT write int main() function
            if (!root){
                return true;
            }

            traversal(root);
            return isPalindrome(trav);
        }
    };
}

// Reverse a single linked list by iteration.
Node* reverseList(Node *root){

    Node *new_next = nullptr;

    while(root){
        auto old_next = root->next;
        root->next = new_next;
        new_next = root;
        root = old_next;
    }
    return new_next;
}

// Reverse a single linked list by recursion.
namespace ReverseLinkedListRecursion {

    Node * reverseListRecursionHelper(Node *rest, Node *reversed){
        if (!rest)
            return reversed;

        Node *next = rest->next;
        rest->next = reversed;
        return reverseListRecursionHelper(next, rest);
    }
```

```
    Node * reverseListRecursion(Node* root){
        return reverseListRecursionHelper(root, nullptr);
    }
}


namespace ReverseRangeOfLinkedList {
    class Solution {
    public:

        // Reverse first k nodes of a linked list.
        ListNode *reverseFirstKNodes(ListNode *head, int k){
            auto headcopy = head;
            for (int i = 0; i < k; i++) {
                headcopy = headcopy->next;
            }
            auto new_next = headcopy;
            headcopy = head;

            for (int i = 0; i < k; i++) {
                ListNode *old_next = headcopy->next;
                headcopy->next = new_next;
                new_next = headcopy;
                headcopy = old_next;
            }
            return new_next;
        }

        // Reverse nodes from m to n. Counts from index 1.
        // Given 1->2->3->4->5->NULL, m = 2 and n = 4,
        // return 1->4->3->2->5->NULL.
        ListNode *reverseBetween(ListNode *head, int m, int n) {
            // Start typing your C/C++ solution below
            // DO NOT write int main() function
            ListNode *headcopy = head;

            int new_m = m - 1;
            int new_n = n - m + 1;

            if (!new_m)
                return reverseFirstKNodes(head, new_n);

            for (int i = 0; i < new_m - 1; i++) {
                headcopy = headcopy->next;
            }

            headcopy->next = reverseFirstKNodes(headcopy->next, new_n);
            return head;
        }
    };
}

//    Given an unsorted integer array, find the first missing positive integer.
//
//    For example,
//    Given [1,2,0] return 3,
//    and [3,4,-1,1] return 2.
//
//    Your algorithm should run in O(n) time and uses constant space.
class Solution {
public:
    int firstMissingPositive(int A[], int n) {
        // Start typing your C/C++ solution below
        // DO NOT write int main() function
        int i = 0;
        while(i < n){
            if (A[i] == i){
            }
            else if (A[i] < 0){
                A[i] = 0;
            }
            else if (A[i] > n){
                A[i] = 0;
            }
            else if (A[i] == n){
                A[0] = A[i];
            }
```

```
            else if (A[i] == A[A[i]]){
                // The corresponding number is already there, excluding comparing
                // with itself since we already checked that case.
                A[i] = 0;
            }
            else {
                swap(A[i], A[A[i]]);
                continue;
            }
            i++;
        }

        for (i = 1; i < n; i++) {
            if (A[i] != i)
                return i;
        }
        if (A[0] == i)
            i++;
        return i;
    }
};


//    Implement next permutation, which rearranges numbers into the
//    lexicographically next greater permutation of numbers.
//
//    If such arrangement is not possible, it must rearrange it as the
//    lowest possible order (ie, sorted in ascending order).
//
//    The replacement must be in-place, do not allocate extra memory.
//
//    Here are some examples. Inputs are in the left-hand column and
//    its corresponding outputs are in the right-hand column.
//
//    1,2,3 → 1,3,2
//    3,2,1 → 1,2,3
//    1,1,5 → 1,5,1
class Solution {
public:
    void nextPermutation(vector<int> &num) {
        // Start typing your C/C++ solution below
        // DO NOT write int main() function
        if (is_sorted(num.rbegin(), num.rend())){
            reverse(num.rbegin(), num.rend());
            return;
        }
        int i, j;
        for(i = (int)num.size() - 2; i > 0; --i){
            if (num[i] < num[i + 1])
                break;
        }
        for (j = (int)num.size() - 1; j > i; --j) {
            if (num[j] > num[i])
                break;
        }
        swap(num[i], num[j]);
        reverse(num.begin() + i + 1, num.end());
    }
};



int main(){

    int n = int(strlen(r));
    r[n] = 0;
    da(n+1, 128);
    for (int i = 0; i < n + 1; i++) {
        printf("%d ", sa[i]);
    } printf("\n");
    calheight(n);
    for (int i = 0; i < n + 1; i++) {
        printf("%d ", height[i]);
    } printf("\n");
    return 0;
}
```