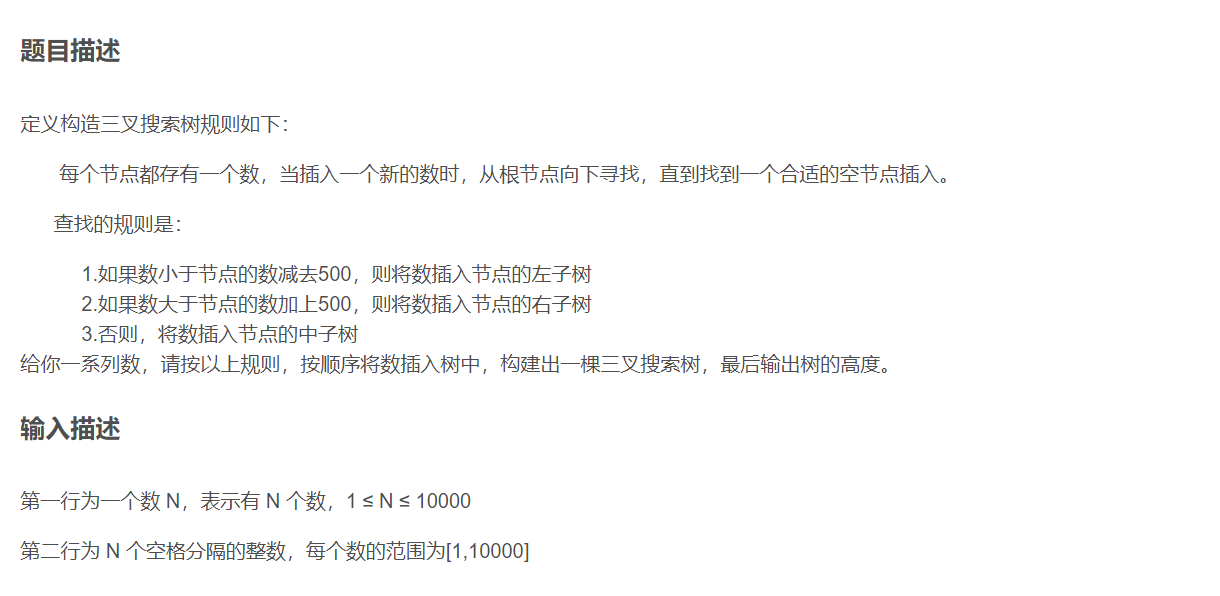
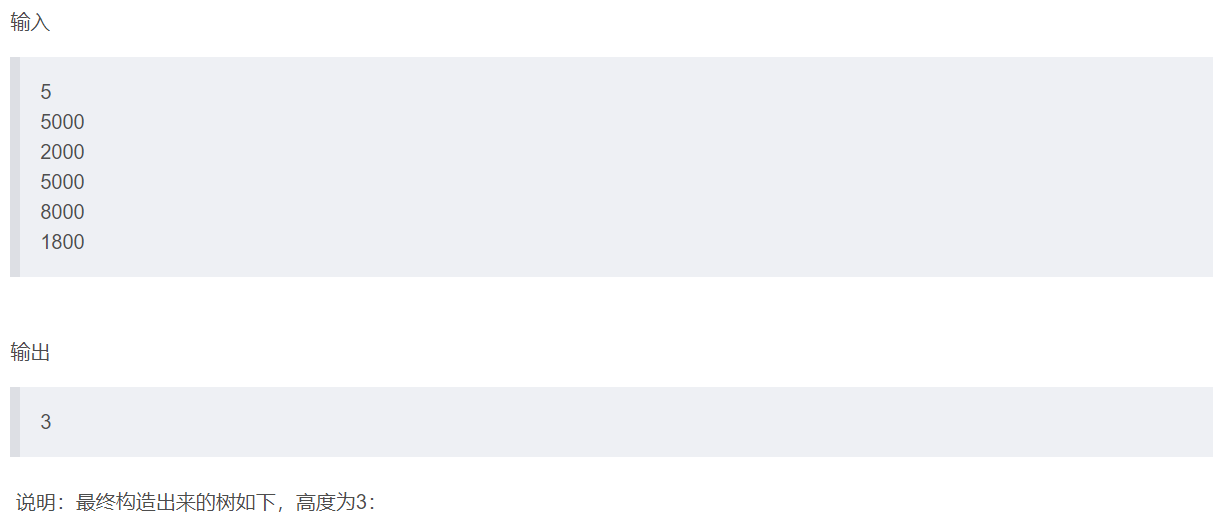
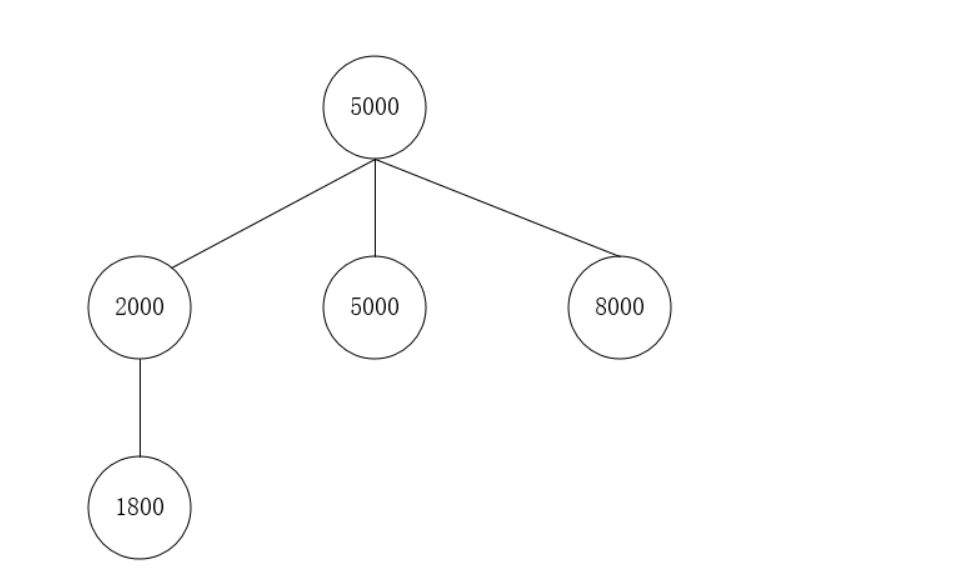
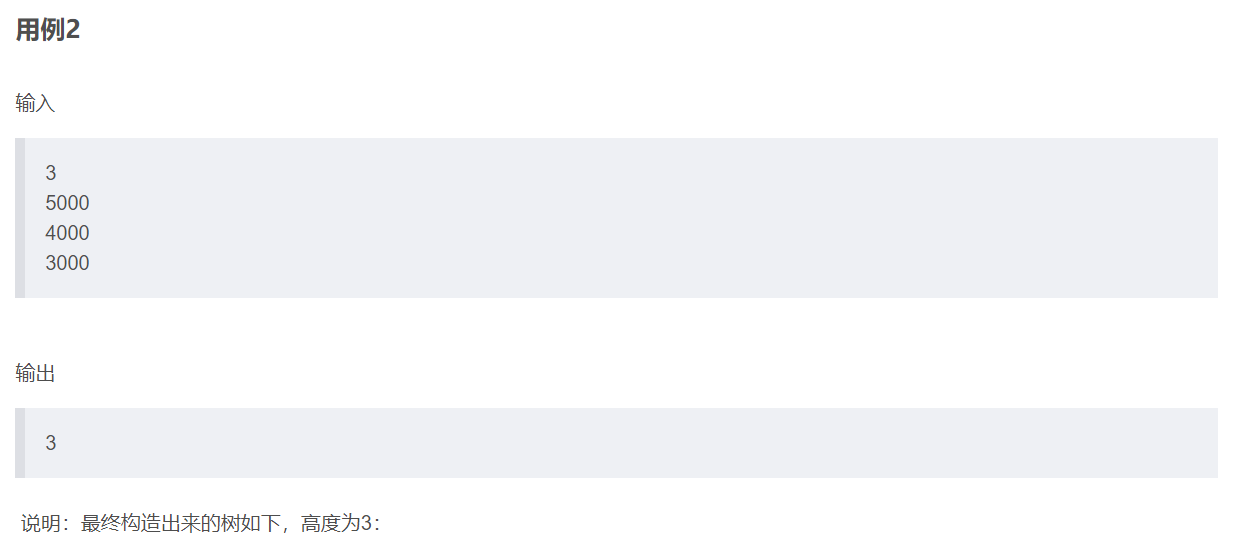
# **E卷-计算三叉搜索树的高度[100分]（ Java | Python3 | C++ | C语言 | JsNode | Go ）**

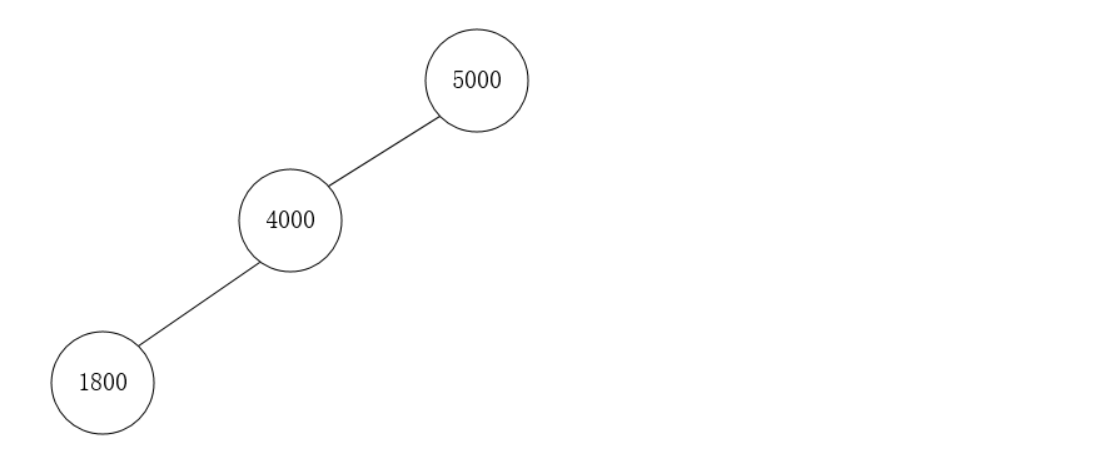














import java.util.Scanner;

public class Main {

static class Node {

int val;

Node left; // 左子树

Node mid; // 中子树

Node right; // 右子树

Node(int v) {

val = v;

left = null; // 初始为空

mid = null; // 初始为空

right = null; // 初始为空

}

}

public static Node build(Node now, int x) {

if (now == null) { // 如果当前节点为空，则创建一个新的节点

return new Node(x);

}

if (x < now.val -

500) { // 如果插入的数小于当前节点的值减去500，则将数插入左子树

now.left = build(now.left, x);

} else if (x > now.val +

500) { // 如果插入的数大于当前节点的值加上500，则将数插入右子树

now.right = build(now.right, x);

} else { // 否则将数插入中子树

now.mid = build(now.mid, x);

}

return now;

}

public static int dfs(Node now) {

if (now == null) { // 如果当前节点为空，则高度为0

return 0;

}

int a = dfs(now.left); // 深度优先遍历左子树

int b = dfs(now.mid); // 深度优先遍历中子树

int c = dfs(now.right); // 深度优先遍历右子树

return 1 + Math.max(a, Math.max(b,

c)); // 当前节点的高度为1，加上子树中的最大高度

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt(); // 读取输入的节点个数

int[] nums = new int[n];

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

nums[i] = sc.nextInt(); // 读取输入的数列

}

Node root = null;

for (int x : nums) {

root = build(root, x); // 按顺序将数插入到三叉树中

}

System.out.println(dfs(root)); // 输出树的高度

}

}



class Node:

def \_\_init\_\_(self, v):

self.val = v

self.left = None # 左子树

self.mid = None # 中子树

self.right = None # 右子树

def build(now, x):

# 如果当前节点为空，则创建一个新的节点

if not now:

return Node(x)

# 如果插入的数小于当前节点的值减去500，则将数插入左子树

if x < now.val - 500:

now.left = build(now.left, x)

# 如果插入的数大于当前节点的值加上500，则将数插入右子树

elif x > now.val + 500:

now.right = build(now.right, x)

# 否则将数插入中子树

else:

now.mid = build(now.mid, x)

return now

def dfs(now):

# 如果当前节点为空，则高度为0

if not now:

return 0

# 深度优先遍历左子树

a = dfs(now.left)

# 深度优先遍历中子树

b = dfs(now.mid)

# 深度优先遍历右子树

c = dfs(now.right)

# 当前节点的高度为1，加上子树中的最大高度

return 1 + max(a, b, c)

# 读取输入的节点个数

n = int(input())

# 读取输入的数列

nums=[]

for \_ in range(n):

x=int(input())

nums.append(x)

root = None

# 按顺序将数插入到三叉树中

for i in nums:

root = build(root, i)

# 输出树的高度

print(dfs(root))



#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Node {

int val;

Node\* left; // 左子树

Node\* mid; // 中子树

Node\* right; // 右子树

Node(int v): val(v), left(nullptr), mid(nullptr),

right(nullptr) {} // 初始化节点

};

Node\* build(Node\* now, int x) {

if (!now) return new Node(

x); // 如果当前节点为空，则创建一个新的节点

if (x < now->val - 500) {

now->left = build(now->left,

x); // 如果插入的数小于当前节点的值减去500，则将数插入左子树

} else if (x > now->val + 500) {

now->right = build(now->right,

x); // 如果插入的数大于当前节点的值加上500，则将数插入右子树

} else {

now->mid = build(now->mid, x); // 否则将数插入中子树

}

return now;

}

int dfs(Node\* now) {

if (!now) return 0; // 如果当前节点为空，则高度为0

int a = dfs(now->left); // 深度优先遍历左子树

int b = dfs(now->mid); // 深度优先遍历中子树

int c = dfs(now->right); // 深度优先遍历右子树

return 1 + max({a, b, c}); // 当前节点的高度为1，加上子树中的最大高度

}

int main() {

int n;

cin >> n; // 读取输入的节点个数

vector<int> nums(n);

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

cin >> nums[i]; // 读取输入的数列

}

Node\* root = nullptr;

for (int x : nums) {

root = build(root, x); // 按顺序将数插入到三叉树中

}

cout << dfs(root) << endl; // 输出树的高度

return 0;

}



#include <stdio.h>

#include <stdlib.h>

// 定义节点的结构

struct Node {

int val; // 节点的数值

struct Node\* left; // 左子树

struct Node\* mid; // 中子树

struct Node\* right; // 右子树

};

// 创建一个新的节点

struct Node\* newNode(int value) {

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

node->val = value;

node->left = NULL;

node->mid = NULL;

node->right = NULL;

return node;

}

// 按照规则插入节点

struct Node\* build(struct Node\* now, int x) {

// 如果当前节点为空，则创建一个新的节点

if (now == NULL) {

return newNode(x);

}

// 如果插入的数小于当前节点的值减去500，则将数插入左子树

if (x < now->val - 500) {

now->left = build(now->left, x);

}

// 如果插入的数大于当前节点的值加上500，则将数插入右子树

else if (x > now->val + 500) {

now->right = build(now->right, x);

}

// 否则将数插入中子树

else {

now->mid = build(now->mid, x);

}

return now;

}

// 计算树的高度

int dfs(struct Node\* now) {

// 如果当前节点为空，则高度为0

if (now == NULL) {

return 0;

}

// 深度优先遍历左子树

int a = dfs(now->left);

// 深度优先遍历中子树

int b = dfs(now->mid);

// 深度优先遍历右子树

int c = dfs(now->right);

// 当前节点的高度为1，加上子树中的最大高度

return 1 + (a > b ? (a > c ? a : c) : (b > c ? b : c));

}

int main() {

int n;

// 读取输入的节点个数

scanf("%d", &n);

// 读取输入的数列

int nums[n];

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

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

}

struct Node\* root = NULL;

// 按顺序将数插入到三叉树中

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

root = build(root, nums[i]);

}

// 输出树的高度

printf("%d\n", dfs(root));

return 0;

}



class Node {

constructor(val) {

this.val = val; // 节点的数值

this.left = null; // 左子树

this.mid = null; // 中子树

this.right = null; // 右子树

}

}

// 按照规则插入节点

function build(now, x) {

// 如果当前节点为空，则创建一个新的节点

if (!now) {

return new Node(x);

}

// 如果插入的数小于当前节点的值减去500，则将数插入左子树

if (x < now.val - 500) {

now.left = build(now.left, x);

}

// 如果插入的数大于当前节点的值加上500，则将数插入右子树

else if (x > now.val + 500) {

now.right = build(now.right, x);

}

// 否则将数插入中子树

else {

now.mid = build(now.mid, x);

}

return now;

}

// 计算树的高度

function dfs(now) {

// 如果当前节点为空，则高度为0

if (!now) {

return 0;

}

// 深度优先遍历左子树

const a = dfs(now.left);

// 深度优先遍历中子树

const b = dfs(now.mid);

// 深度优先遍历右子树

const c = dfs(now.right);

// 当前节点的高度为1，加上子树中的最大高度

return 1 + Math.max(a, b, c);

}

// 读取输入

const readline = require("readline").createInterface({

input: process.stdin,

output: process.stdout,

});

let nums = [];

readline.question("", (n) => {

const numCount = parseInt(n);

let count = 0;

readline.on("line", (line) => {

nums.push(parseInt(line));

count++;

if (count === numCount) {

readline.close();

let root = null;

// 按顺序将数插入到三叉树中

for (let i of nums) {

root = build(root, i);

}

// 输出树的高度

console.log(dfs(root));

}

});

});



package main

import (

"bufio"

"fmt"

"os"

"strconv"

)

// 节点结构体

type Node struct {

val int // 节点的值

left \*Node // 左子树

mid \*Node // 中子树

right \*Node // 右子树

}

// 创建一个新的节点

func newNode(value int) \*Node {

// 分配内存并初始化节点的值

return &Node{val: value}

}

// 根据规则构建树

func build(now \*Node, x int) \*Node {

// 如果当前节点为空，则创建一个新的节点

if now == nil {

return newNode(x)

}

// 如果插入的值小于当前节点的值减去500，则插入左子树

if x < now.val-500 {

now.left = build(now.left, x)

// 如果插入的值大于当前节点的值加上500，则插入右子树

} else if x > now.val+500 {

now.right = build(now.right, x)

// 否则插入中子树

} else {

now.mid = build(now.mid, x)

}

// 返回当前节点

return now

}

// 深度优先搜索计算树的高度

func dfs(now \*Node) int {

// 如果当前节点为空，则高度为0

if now == nil {

return 0

}

// 深度优先遍历左子树

a := dfs(now.left)

// 深度优先遍历中子树

b := dfs(now.mid)

// 深度优先遍历右子树

c := dfs(now.right)

// 当前节点的高度为1，加上子树中的最大高度

return 1 + max(a, b, c)

}

// 返回三个数字中的最大值

func max(a, b, c int) int {

if a > b {

if a > c {

return a

}

return c

}

if b > c {

return b

}

return c

}

func main() {

// 读取输入

scanner := bufio.NewScanner(os.Stdin)

// 读取节点个数

scanner.Scan()

n, \_ := strconv.Atoi(scanner.Text())

nums := make([]int, n)

// 读取节点值并存储到数组中

for i := 0; i < n; i++ {

scanner.Scan()

nums[i], \_ = strconv.Atoi(scanner.Text())

}

var root \*Node

// 按顺序将数插入到三叉树中

for \_, num := range nums {

root = build(root, num)

}

// 输出树的高度

fmt.Println(dfs(root))

}