12: Next Permutation

24: Single Number II

32: Swap Nodes in Pairs

37: Reorder List

62: Binary Tree Level Order Traversal II

67: Balanced Binary Tree

88: First Missing Positive

127: Scramble String

129: Edit Distance

140: Multiply Strings

--------

**2.1.12 Next Permutation**

描述

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

void next\_permutation(string& input)

{

// for STL, just call ‘next\_permutation(input.begin(), input.end())’

// my own implementation for string goes below

// handle boundaries

int len = input.length();

if (len<=1) return;

// 3 steps in total, when string length is at least 2

// step 1 – find the rightmost char that’s strictly less than its right neighbor

int first = len-2;

for (; input[first]>=input[first+1];first--]) {

if (first==0) {

reverse(input.begin(), input.end()); return;

}

}

int mid = first+1;

int last = len-1;

for (; input[last]<=input[first]; last--) ;

swap(input[first], input[last]);

reverse(input.begin()+mid, input.end());

}

**2.1.24 Single Number II**

描述

Given an array of integers, every element appears three times except for one. Find that single one.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using

extra memory?

// method: take modular 3 for sum of every bit position

// I would use one variable to store the result.

// In case I am not allowed to use one variable, I can do this: check if the first one is the element (in linear time) to be found. If it is, then we have the solution. If not, we set

vector<int> data = { 4,4,5,4,5,2,5,2,7,2 };

void mod3(int& a, vector<int>& result)

{

for (int i=0; i<32; i++) {

result[i] = (((a>>i)&1) + result[i]) % 3;

}

}

bool findIt(vector<int>& data)

{

size\_t sz = data.size();

if (sz<4 || sz%3 != 1)

return false;

vector<int> result(32,0);

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

mod3(data[i], result);

}

int theOne = 0;

for (int i=0; i<32; i++) {

theOne += result[i] << i;

}

cout << theOne;

}

**2.2.8 Swap Nodes in Pairs**

描述

Given a linked list, swap every two adjacent nodes and return its head.

For example, Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes

itself can be changed.

// 1. define node

struct Node {

int data;

Node\* next;

Node(int d) :data(d), next(nullptr) { }

};

// 2. play with pointers to swap nodes

Node\* swapEveryTwoInList(Node\* head)

{

// boundary

if (!head || !head->next)

return head;

Node\* newHead = head->next;

// general

Node\* p = head;

Node\* q = p->next;

while (p && q) {

// change q’s next pointers

Node\* tmp = p->next->next;

q->next = p;

// now what’s p’s next pointer?

if (!tmp || !tmp->next) { p->next = tmp; break; }

p->next = tmp->next;

p = tmp;

q = tmp->next;

}

return newHead;

}

**2.2.13 Reorder List**

描述

Given a singly linked list *L* : *L*0 *! L*1 *! \_ \_ \_ ! Ln􀀀*1 *! Ln*, reorder it to: *L*0 *! Ln ! L*1 *!*

*Ln􀀀*1 *! L*2 *! Ln􀀀*2 *! \_ \_ \_*

You must do this in-place without altering the nodes’ values.

For example, Given {1,2,3,4}, reorder it to {1,4,2,3}.

struct Node

{

int data;

Node\* next;

Node(int d):data(d),next(nullptr) {}

};

void reorderList(Node\* head)

{

// boundary

if (!head || !head->next) return head;

// 1. get the odd-indexed sub-list

Node\* odd = head;

Node\* p = odd;

while (p) {

if (!p->next) break;

p->next = p->next->next;

p=p->next;

}

// 2. get the even-indexed sub-list

Node\* even = head->next;

p = even;

while (p) {

if (!p->next) break;

p->next = p->next->next;

p = p->next;

}

// 3. reverse the even-indexed sub-list

// 4. join the two list

}

**5.1.5 Binary Tree Level Order Traversal II**

描述

Given a binary tree, return the bottom-up level order traversal of its nodes’ values. (ie, from left to right,

level by level from leaf to root).

For example: Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

5.1 二叉树的遍历95

[

[15,7]

[9,20],

[3],

]

// The easier way to do level-order is to do DFS while remembering maintaining level information, we push\_back Level N’s elements to its dedicated result queue/vector.

struct Node {

int data;

Node\* left;

Node\* right;

Node(int d) : data(d), left(nullptr), right(nullptr);

};

void dfs(Node\* parent, vector<vector<int> >& result, int level) {

if (!parent) return;

if (level==result.size()) result->push\_back\*(vector<int>());

result[level].push\_back(parent->data);

dfs(parent->left, result, level+1);

dfs(parent->right, result, level+1);

}

void bottomUpLevelOrder(Node\* root) {

vector<vector<int> > result;

dfs(root, result, 0);

reverse(result.begin(), result.end());

}

**5.1.10 Balanced Binary Tree**

描述

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two

subtrees of every node never differ by more than 1.

struct Node {

int data;

Node\* left;

Node\* right;

Node(int d) :data(d), left(nullptr), right(nullptr) {}

};

void dfs(Node\* parent, set<int>& depths, int depth)

{

if (!parent) return;

if (!parent->left && !parent->right) { depths.insert(depth); return; }

if (parent->left)

dfs(parent->left, depths, depth + 1);

if (parent->right)

dfs(parent->right, depths, depth + 1);

}

bool isBalanced(Node\* root)

{

if (!root) return true;

set<int> depths;

dfs(root, depths, 0);

if ( depths.size() <= 1 || (abs(\*depths.begin() - \*(prev(depths.end())))<=1) )

return true;

else

return false;

}

**6.6 First Missing Positive**

描述

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.

//

int firstMissingPositive(vector<int>& data)

{

int sz = data.size();

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

int d = data[i]; // should be from 1 to sz

while (d>=1 && d<=sz && d!=i+1) {

swap(data[i], data[d-1]);

d = data[i];

}

}

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

if (data[i]!=(i+1)) {

cout << i+1 << “ is missing” << endl;

break;

}

}

}

**13.7 Scramble String**

描述

Given a string *s*1, we may represent it as a binary tree by partitioning it to two non-empty substrings

recursively.

Below is one possible representation of s1 = ”great”:

great

/ \

gr eat

/ \ / \

g r e at

/ \

a t

204 第13 章动态规划

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node ”gr” and swap its two children, it produces a scrambled string

”rgeat”.

rgeat

/ \

rg eat

/ \ / \

r g e at

/ \

a t

We say that ”rgeat” is a scrambled string of ”great”.

Similarly, if we continue to swap the children of nodes ”eat” and ”at”, it produces a scrambled string

”rgtae”.

rgtae

/ \

rg tae

/ \ / \

r g ta e

/ \

t a

We say that ”rgtae” is a scrambled string of ”great”.

Given two strings *s*1 and *s*2 of the same length, determine if *s*2 is a scrambled string of *s*1.

//Let me just use recursion for the time being

bool isScramble(string& s1, string& s2, int start1, int end1, start2)

{

size\_t len = end1 – start1 - 1; // end is 1 beyond the last

if (len==0) return true;

if (len==1) { if (s1[start1]==s2[start2]) return true; else return false }

int end2 = start2 + (end1 – start1);

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

if ( (isScrameble(s1, s2, start1, start1+i+1, start2) && isScramble(s1, s2, start1+i+1, end1, start2+i+1)) || (isScrameble(s1, s2, start1, start1+i+1, end2-i) && isScramble(s1, s2, start1+i, end1, start2)) ) return true;

}

}

**Edit Distance**

描述

Given two words word1 and word2, find the minimum number of steps required to convert word1 to

word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

13.9 Edit Distance 211

• Insert a character

• Delete a character

• Replace a character

// Dynamic programming, very interesting analysis

int minDistance(const string& w1, const string& w2) {

const size\_t n = w1.size();

const size\_t m = w2.size();

int f[n+1][m+1];

for (size\_t j = 0; j<m+1; j++) f[0][j]=j;

for (size\_t i = 0; i<n+1, i++) f[i][0]=i;

for (size\_t i=1; i<=n; i++) {

for (size\_t j=1; j<=m; j++) {

if (w1[i-1] = w2[j-1]) f[i][j] = f[i-1][j-1];

else {

int mn = min(f[i-1][j], f[i][j-1]);

f[i][j] = 1 + min(f[i-1][j-1], mn)

}

}

return f[n][m];

}

**15.6 Multiply Strings**

描述

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.