Algorithms

[STL 3](#_Toc425858149)

[Vector 3](#_Toc425858150)

[String 3](#_Toc425858151)

[Covert to string 3](#_Toc425858152)

[Sort 3](#_Toc425858153)

[Queue 3](#_Toc425858154)

[Reverse 3](#_Toc425858155)

[unordered\_map 3](#_Toc425858156)

[unordered\_set 3](#_Toc425858157)

[<climits> 3](#_Toc425858158)

[Bit Operations 4](#_Toc425858159)

[priority\_queue 4](#_Toc425858160)

[<cmath> <math.h> 4](#_Toc425858161)

[C 4](#_Toc425858162)

[Initialize arrays 4](#_Toc425858163)

[C++ 4](#_Toc425858164)

[Class Constructor and Destructor 4](#_Toc425858165)

[enum 4](#_Toc425858166)

[Summarizes the sort algorithms 4](#_Toc425858167)

[Frequently Asked Questions 5](#_Toc425858168)

[►Word Break 5](#_Toc425858169)

[Remove Duplicates from Sorted Array II 5](#_Toc425858170)

[►Sort List (tricky “Sentinel value” and “Assistant Counting”) 5](#_Toc425858171)

[►Maximum Product Subarray 6](#_Toc425858172)

[►Find Minimum in Rotated Sorted Array 6](#_Toc425858173)

[►Reverse Words in a String 7](#_Toc425858174)

[Path Sum II 7](#_Toc425858175)

[Minimum Depth of Binary Tree 7](#_Toc425858176)

[Balanced Binary Tree 8](#_Toc425858177)

[Search a 2D Matrix 8](#_Toc425858178)

[Set Matrix Zeroes 9](#_Toc425858179)

[Linked List 9](#_Toc425858180)

[►Convert Sorted List to Binary Search Tree 9](#_Toc425858181)

[Convert Sorted Array to Binary Search Tree 9](#_Toc425858182)

[Binary Tree or Binary Search Tree 10](#_Toc425858183)

[►Binary Tree Inorder Traversal (Non Recursive) 10](#_Toc425858184)

[Validate Binary Search Tree 10](#_Toc425858185)

[How to handle duplicates in Binary Search Tree? 10](#_Toc425858186)

[Flatten Binary Tree to Linked List 10](#_Toc425858187)

[Construct Binary Tree from Inorder and Postorder Traversal 11](#_Toc425858188)

[►►Count Complete Tree Nodes 11](#_Toc425858189)

[Graph algorithms 11](#_Toc425858190)

[Breadth-First-Search(Shortest Path) 11](#_Toc425858191)

[►Binary Tree Zigzag Level Order Traversal 11](#_Toc425858192)

[►Binary Tree Right Side View(牢记：带level识别的宽度优先算法) 11](#_Toc425858193)

[►Word Ladder 12](#_Toc425858194)

[Depth-first Search 13](#_Toc425858195)

[►**Course Schedule** 13](#_Toc425858196)

[►Number of Islands 13](#_Toc425858197)

[Surrounded Regions 13](#_Toc425858198)

[►Course Schedule II ---- Toplogical-Sort 14](#_Toc425858199)

[Back Tracking 14](#_Toc425858200)

[►►N-Queens (经典) 14](#_Toc425858201)

[►Combination Sum (背：组合和满足target,元素可重复 15](#_Toc425858202)

[►Combination Sum II(背：组合和满足target，元素不可重复 15](#_Toc425858203)

[► Palindrome Partitioning 16](#_Toc425858204)

[Restore IP Addresses 16](#_Toc425858205)

[Word Search in Maze 16](#_Toc425858206)

[Combinations 17](#_Toc425858207)

[►►►Sudoku Solver 17](#_Toc425858208)

[Dynamic Programming 18](#_Toc425858209)

[►Unique Binary Search Trees 18](#_Toc425858210)

[►House Robber 18](#_Toc425858211)

[找小偷问题 18](#_Toc425858212)

[►House Robber II 19](#_Toc425858213)

[►Dungeon Game(地牢) 19](#_Toc425858214)

[►Decode Ways 19](#_Toc425858215)

[►Longest Substring Without Repeating Characters 20](#_Toc425858216)

[Minimum Path Sum 20](#_Toc425858217)

[Unique Paths 20](#_Toc425858218)

[Unique Paths II 21](#_Toc425858219)

[►►Maximum Subarray 21](#_Toc425858220)

[Maximal Square 21](#_Toc425858221)

[►►►Edit Distance 21](#_Toc425858222)

[Greedy 22](#_Toc425858223)

[►►Jump Game II 22](#_Toc425858224)

[►Gas Station 22](#_Toc425858225)

[Puzzles & Math 23](#_Toc425858226)

[Subsets II 23](#_Toc425858227)

[►Divide Two Integers 23](#_Toc425858228)

[Rotate Array 23](#_Toc425858229)

[►Fraction to Recurring Decimal 24](#_Toc425858230)

[Factorial Trailing Zeroes 24](#_Toc425858231)

[►Maximum Gap 24](#_Toc425858232)

[►3 Sum 25](#_Toc425858233)

[►Permutations 25](#_Toc425858234)

[►►Permutations II 25](#_Toc425858235)

[►►Permutation Sequence 26](#_Toc425858236)

[Multiply Strings 26](#_Toc425858237)

[►Pow(x, n) 27](#_Toc425858238)

[►Majority Element 27](#_Toc425858239)

[►►Majority Element II 27](#_Toc425858240)

[►►►Valid Number (DFS , Regular Expression) 27](#_Toc425858241)

[Best Time to Buy and Sell Stock 28](#_Toc425858242)

[Best Time to Buy and Sell Stock I 28](#_Toc425858243)

[Best Time to Buy and Sell Stock II 28](#_Toc425858244)

[Two Pointers, Sliding Window 29](#_Toc425858245)

[Container With Most Water 29](#_Toc425858246)

[►Minimum Size Subarray Sum 29](#_Toc425858247)

[►Sort Colors 29](#_Toc425858248)

[►►►Trapping Rain Water 29](#_Toc425858249)

[►►►Minimum Window Substring 29](#_Toc425858250)

[Stack 30](#_Toc425858251)

[►►►Largest Rectangle in Histogram 30](#_Toc425858252)

[Thinking Recursively 30](#_Toc425858253)

[►Search in Rotated Sorted Array 30](#_Toc425858254)

[Search in Rotated Sorted Array II 31](#_Toc425858255)

[Search for a Range 31](#_Toc425858256)

[►Unique Binary Search Trees II 31](#_Toc425858257)

[Symmetric Tree 31](#_Toc425858258)

[►LCA -- Lowest Common Ancestor in a Binary Tree 32](#_Toc425858259)

[Spiral Matrix 34](#_Toc425858260)

[Spiral Matrix II 34](#_Toc425858261)

[Rotate Image 35](#_Toc425858262)

[String ----Trie 35](#_Toc425858263)

[►Implement Trie (Prefix Tree) 35](#_Toc425858264)

[►Add and Search Word - Data structure design 35](#_Toc425858265)

[String 36](#_Toc425858266)

[►►►Wildcard Matching 36](#_Toc425858267)

[►Longest Palindromic Substring 37](#_Toc425858268)

[Basic Calculator 37](#_Toc425858269)

[►Basic Calculator II 37](#_Toc425858270)

[Evaluate Reverse Polish Notation 38](#_Toc425858271)

[Simplify Path 38](#_Toc425858272)

[►►►String ----Rolling Hash 38](#_Toc425858273)

[►►►Text Justification 39](#_Toc425858274)

[Heap 39](#_Toc425858275)

[Kth Largest Element in an Array 39](#_Toc425858276)

[Hash 39](#_Toc425858277)

[►Anagrams 39](#_Toc425858278)

[►Longest Sequence in Array 40](#_Toc425858279)

[►Minimum Interval in N Sorted List 40](#_Toc425858280)

[English 41](#_Toc425858281)

[Describe Merge Sorted List Algorithm 41](#_Toc425858282)

[Facebook 面经 41](#_Toc425858283)

[isDistanceZeroOrOne 41](#_Toc425858284)

[Google 面经 41](#_Toc425858285)

[Next Prime Number 41](#_Toc425858286)

[New Random() 41](#_Toc425858287)

[suppose you have a cluster, and each machine in this cluster has a large number of numbers. How can you find out the median of all the numbers on all the machines. Ring 41](#_Toc425858288)

[AirBNB 面经 41](#_Toc425858289)

[►CoinChange.java 41](#_Toc425858290)

[►LowerUpperCasePermutation.java 42](#_Toc425858291)

[►KEditDistance.java 42](#_Toc425858292)

[►ParseCSV.java 42](#_Toc425858293)

[►Bitwise AND of Numbers Range 43](#_Toc425858294)

[Unclassified 43](#_Toc425858295)

[►►Insert Interval 43](#_Toc425858296)

[►►►First Missing Positive 43](#_Toc425858297)

[Reservoir Sampling 43](#_Toc425858298)

# STL

## Vector

vector<int> v(10); vector<double> v(10, 8.9);

push\_back() pop\_back() back() front()

clear() : Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.

int idx = 10;

digits.erase(digits.begin() + idx);

vector<vector<int> > result;

//initialize vector with array

static const int arr[] = {16,2,77,29};

vector<int> vec (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int> v { 34,23 };

// or

std::vector<int> v = { 34,23 };

//initialize 2d vector

vector<vector<int>> matrix(n,vector<int> (n, 0));

## String

string += str more efficient

substr (size\_type pos = 0, size\_type len = npos) const;

const charT\* c\_str() const;

str.length()

sol.resize(sol.length() - 1); s.compare(word)==0 // string equal

## Covert to string

#include <string>

string str = to\_string(40);

Convert c style string to int, first check string is “0”?

#include <stdlib.h> /\* atoi \*/

int atoi (const char \* str);

return 0: faild

atof Convert string to double (function )

atoi Convert string to integer (function )

atol Convert string to long integer (function )

atoll Convert string to long long integer (function )

strto Convert string to double (function )

strtof Convert string to float (function )

strtol Convert string to long integer (function )

## Sort

Sort String

sort(strs.begin(), strs.end(), myComp);

static bool myComp(string a, string b) { … }

Sort Int

default **ascending, dscending** use >:

struct compare{

bool operator()(const int& l, const int& r) {

return l > r;

}

};

or use

#include <functional> // greater

sort(strs.begin(), strs.end(), greater<int>() );

Sort Object

//**ascending** use <, **dscending** use >:

static bool comp(const Interval& a,const Interval& b){

return a.start < b.start;

}

sort(intervals.begin(), intervals.end(), comp);

## Queue

/\*Elements are pushed into the "back" of the specific container and popped from its "front".\*/

back() emplace() empty() front() pop() push() size() swap()

## Reverse

vector<int> result; reverse(result.begin(), result.end());

## unordered\_map

unordered\_map<string,bool> hash; hash.count(pre)<=0 hash[post] = false;

hash[i]++; // no need to init

unordered\_map<int,int>::iterator it;

for(it=m.begin();it!=m.end();it++){ it->first; it->second; }

for (auto& kv : myMap) {

cout << kv.first << " has value " << kv.second <<endl;

}

## unordered\_set

unordered\_set<string> dict; dict.insert(str); dict.count(s)>0

## <climits>

|  |  |  |
| --- | --- | --- |
| INT\_MIN | Minimum value for an object of type int | -32767 (-215+1) or less\* |
| INT\_MAX | Maximum value for an object of type int | 32767 (215-1) or greater\* |
| UINT\_MAX | Maximum value for an object of type unsigned int | 65535 (216-1) or greater\* |
| LONG\_MIN | Minimum value for an object of type long int | -2147483647 (-231+1) or less\* |
| LONG\_MAX | Maximum value for an object of type long int | 2147483647 (231-1) or greater\* |

## Bit Operations

b = b<<1;

The Bitwise Complement: ~, flips every bit.

## priority\_queue

push() pop() top() empty() size()

default **max heap, min heap** use >:

struct compare {

bool operator()(const int& l, const int& r) {

return l > r;

}

};

or use

#include <functional> // greater

priority\_queue<int, vector<int>, greater<int> > min\_queue;

// constructing priority queues

#include <iostream> // cout

#include <queue> // priority\_queue

#include <vector> // vector

using namespace std;

class mycomparison

{

bool reverse;

public:

mycomparison(const bool& revparam=false)

{reverse=revparam;}

bool operator() (const int& lhs, const int&rhs) const

{

if (reverse) return (lhs>rhs);

else return (lhs<rhs);

}

};

int main ()

{

int myints[]= {10,60,50,20};

priority\_queue<int> first;

priority\_queue<int> second (myints,myints+4);

priority\_queue<int, vector<int>, greater<int> >

third (myints,myints+4);

// using mycomparison:

typedef priority\_queue<int,vector<int>,mycomparison> mypq\_type;

mypq\_type fourth; // less-than comparison

mypq\_type fifth (mycomparison(true)); // greater-than comparison

return 0;

}

## <cmath> <math.h>

pow(7.0, 3.0);

Try bit left shift like return (2 << x) to compute power instead of pow(2, x)

**Tip: 演算很重要！**

**Tip: Sometimes write down the input & output makes you find the pattern quickly**

# C

## Initialize arrays

bool row[9][10];

bool column[9][10];

bool cube[3][3][10];

memset(row, 0, sizeof(row));

memset(col, 0, sizeof(col));

memset(cube, 0, sizeof(cube));

# C++

## Class Constructor and Destructor

#include <iostream>

using namespace std;

class Line

{

public:

void setLength( double len );

double getLength( void );

Line(); // This is the constructor declaration

Line( double len): length(len) {

cout << "Object is being created" << endl;

}

private:

double length;

};

Line::~Line() {

cout << "Object is being deleted" << endl;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main( )

{

Line line(10.0);

cout << "Length of line : " << line.getLength() <<endl;

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

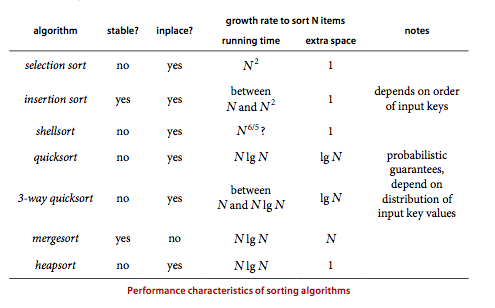
别托大，演算很重要！！

## enum

enum Color{White, Gray, Black};

vector<Color> color(numCourses, White);

# Summarizes the sort algorithms



# Frequently Asked Questions

## ►Word Break

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

For example, given  
*s* = "leetcode",  
*dict* = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

Runtime: **64 ms**

class Solution {

public:

bool debug;

unordered\_map<string,bool> presub;

bool wordBreak(string s, unordered\_set<string> &dict) {

debug = false;

if(debug) cout<<s<<endl;

if(dict.count(s)>0) {

if(debug) cout<<s<<" true"<<endl;

return true;

}

for(int i=1; i<s.size(); i++){

string pre = s.substr(0,i);

string post = s.substr(i,s.size()-i);

bool flagpre = false;

bool flagpost = false;

if(presub.count(pre)<=0) flagpre = wordBreak( pre, dict);

else flagpre = presub[pre];

if(flagpre){

if(presub.count(post)<=0) flagpost = wordBreak( post, dict);

else flagpost = presub[post];

if(flagpost){

if(debug) cout<<s<<" true"<<endl;

presub[s] = true;

return true;

}

}

}

if(debug) cout<<s<<" false"<<endl;

presub[s] = false;

return false;

}

};

**Worst case: O(n^2)**

**Solution2: Check from left to right is not a good idea, I think the best idea is to check from right to left. In addition, we should calculate the effective length of the substring, here is my 0ms DP solution**

class Solution {

public:

bool wordBreak(string s, unordered\_set<string> &dict) {

if (dict.empty())

return false;

int len = s.size(), max\_len = dict.begin()->size(), min\_len = max\_len;

for (auto it = dict.begin(); it != dict.end(); ++it){

if (it->size() > max\_len)

max\_len = it->size();

else if (it->size() < min\_len)

min\_len = it->size();

}

vector<int> flag(len + 1);

flag[len] = 1;

for (int i = len - 1; i >= 0; --i)

for (int j = min\_len; j <= min(max\_len, len - i); ++j)

if (flag[i + j] && dict.find(s.substr(i, j)) != dict.end()) {

flag[i] = 1;

break;

}

return flag[0] == 1;

}

};

## Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":  
What if duplicates are allowed at most *twice*?

For example,  
Given sorted array *nums* = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of *nums* being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.

Tricky:

*nums* = [1,1,1,2,2,3]

equal = [ 1,1,0,1,0]

*flag* = [ 0,1,1,0,1]

equal? flag?

0 0 move, flag=false;

0 1 move, flag=false;

1 0 move, flag=true;

1 1 d++; ret--;

**class** Solution {

**public**:

**int** removeDuplicates(vector<**int**>& nums) {

**int** i, d=0;

**int** n = nums.size();

**int** ret = n;

**bool** flag = **false**;

**if**(n<=1) **return** n;

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

**bool** eq = nums[i]==nums[i-1];

**if**( !eq || !flag ){

**if**(d!=0) nums[i-d] = nums[i];

**if**(eq) flag = **true**;

**else** flag = **false**;

}**else** {

d++;

ret--;

}

}

**return** ret;

}

};

## ►Sort List (tricky “Sentinel value” and “Assistant Counting”)

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

**O(n log n) time, O(1) space solution**

I use a non-recurisve way to write merge sort. For example, the size of ListNode is 8,

Round #1 block\_size = 1

(a1, a2), (a3, a4), (a5, a6), (a7, a8)

Compare a1 with a2, a3 with a4 ...

Round #2 block\_size = 2

(a1, a2, a3, a4), (a5, a6, a7, a8)

merge two sorted arrays (a1, a2) and (a3, a4), then merge tow sorted arrays(a5, a6) and (a7, a8)

Round #3 block\_size = 4

(a1, a2, a3, a4, a5, a6, a7, a8)

merge two sorted arrays (a1, a2, a3, a4), and (a5, a6, a7, a8)

No need for round #4 cause block\_size = 8 >= n = 8

class Solution {

public:

int count\_size(ListNode \*node){ ……

}

ListNode \*sortList(ListNode \*head) {

int block\_size = 1, n = count\_size(head), iter = 0, i = 0, a = 0, b = 0;

ListNode virtual\_head(0);

ListNode \*last = NULL, \*it = NULL, \*A = NULL, \*B = NULL, \*tmp = NULL;

virtual\_head.next = head;

while (block\_size < n){

iter = 0;

last = &virtual\_head;

it = virtual\_head.next;

while (iter < n){

a = min(n - iter, block\_size);

b = min(n - iter - a, block\_size);

A = it;

if (b != 0){

for (i = 0; i < a - 1; ++i) it = it->next;

B = it->next;

it->next = NULL;

it = B;

for (i = 0; i < b - 1; ++i) it = it->next;

tmp = it->next;

it->next = NULL;

it = tmp;

}

while (A || B){

if (B == NULL || (A != NULL && A->val <= B->val)){

last->next = A;

last = last->next;

A = A->next;

} else {

last->next = B;

last = last->next;

B = B->next;

}

}

last->next = NULL;

iter += a + b;

}

block\_size <<= 1;

}

return virtual\_head.next;

}

};

## ►Maximum Product Subarray

Question Solution

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4],  
the contiguous subarray [2,3] has the largest product = 6.

**Lemma: every max product sub array must be start from A[0] xor A[i]==0 or end at the end of A xor A[j]==0**

class Solution {

public:

int maxProduct(int A[], int n) {

int maxP = 0;

int P = 1;

int i;

if(n<=0) return 0;

if(n==1) return A[0];

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

if(A[i] == 0) {

P = 1;

continue;

}

P \*= A[i];

if(P > maxP) maxP = P;

}

P = 1;

for(i=n-1; i>=0; i--){

if(A[i] == 0) {

P = 1;

continue;

}

P \*= A[i];

if(P > maxP) maxP = P;

}

return maxP;

}

};

Runtime: **16 ms**

**Improve Clue: back ward max product means min prodect forward**

class Solution {

public:

int maxProduct(int A[], int n) {

if (n == 0) {

return 0;

}

int maxherepre = A[0];

int minherepre = A[0];

int maxsofar = A[0];

int maxhere, minhere;

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

maxhere = Max(maxherepre \* A[i], minherepre \* A[i], A[i]);

minhere = Min(maxherepre \* A[i], minherepre \* A[i], A[i]);

maxsofar = max(maxhere, maxsofar);

maxherepre = maxhere;

minherepre = minhere;

}

return maxsofar;

}

int Max(int a, int b, int c){

int max = a;

if(max < b) max = b;

if(max < c) max = c;

return max;

}

int Max(int a, int b){

return a>b? a:b;

}

int Min(int a, int b, int c){

int min = a;

if(min > b) min = b;

if(min > c) min = c;

return min;

}

};

## ►Find Minimum in Rotated Sorted Array

Question Solution

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

Runtime: **44 ms**

class Solution {

public:

int findMin(vector<int> &num) {

int i,j,m,mid ;

i = 0;

j = num.size()-1;

while(i<j){

if(i == j - 1 ) return num[i]<num[j]? num[i]:num[j];

if(num[i]<num[j]){

return num[i];

}

m = (i+j)/2;

mid = num[m];

if(num[i] < mid){

i = m;

}else{

j = m;

}

}

return num[i];

}

};

## ►[Reverse Words in a String](https://oj.leetcode.com/problems/reverse-words-in-a-string/)

Given an input string, reverse the string word by word.

For example,  
Given s = "the sky is blue",  
return "blue is sky the".

Runtime: **44 ms**

class Solution {

public:

void reverseWords(string &s) {

string result;

int pos = 0;

for (int i = 0; i < s.size(); i ++){

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

if (i > pos )

result = s.substr(pos,i-pos)+ " " + result ;

pos = i + 1;

}

else if (i == s.size()-1)

result = s.substr(pos,s.size()-pos)+" "+result;

}

s = result.substr(0,result.size()-1) ;

}

};

Runtime: **16 ms**

**Improvement Clue: string += str if faster than string = str + string**

class Solution {

public:

void reverseWords(string &s) {

string result;

int pos = s.size()-1;

for (int i=s.size()-1; i>=0; i--){

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

if (i < pos )

result +=s.substr(i+1,pos-i) + " ";

pos = i - 1;

}

else if (i == 0)

result += s.substr(0, pos+1) + " "; //in special case , we can make the output as the same as imspecial one,

//so than we can handle it uniformlly

}

s = result.substr(0,result.size()-1);

}

};

## Path Sum II

Question Solution

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:  
Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ / \

7 2 5 1

return

[

[5,4,11,2],

[5,8,4,5]

]

Top of Form

Bottom of Form

Runtime: **36 ms**

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int minDepth(TreeNode \*root) {

queue<TreeNode \*> q;

TreeNode \*nextLevel;

int depth = 0;

bool flag = false; //change level flag

if(root != NULL) {

q.push( root );

depth = 1;

if(root->left != NULL) nextLevel = root->left;

else nextLevel = root->right;

}

while(q.size() != 0){

TreeNode \*p = q.front();

q.pop();

if(nextLevel == p){

depth ++;

flag = true;

}

if(p->left == NULL && p->right == NULL){

return depth;

}

if(p->left != NULL) {

q.push( p->left );

if(flag) {

nextLevel = p->left;

flag = false;

}

}

if(p->right != NULL) {

q.push( p->right );

if(flag){

nextLevel = p->right;

flag = false;

}

}

}

return depth;

}

};

## Minimum Depth of Binary Tree

Question Solution

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Runtime: **72 ms**

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int minDepth(TreeNode \*root) {

queue<TreeNode \*> q;

TreeNode \*nextLevel;

int depth = 0;

bool flag = false; //change level flag

if(root != NULL) {

q.push( root );

depth = 1;

if(root->left != NULL) nextLevel = root->left;

else nextLevel = root->right;

}

while(q.size() != 0){

TreeNode \*p = q.front();

q.pop();

if(nextLevel == p){

depth ++;

flag = true;

}

if(p->left == NULL && p->right == NULL){

return depth;

}

if(p->left != NULL) {

q.push( p->left );

if(flag) {

nextLevel = p->left;

flag = false;

}

}

if(p->right != NULL) {

q.push( p->right );

if(flag){

nextLevel = p->right;

flag = false;

}

}

}

return depth;

}

};

## Balanced Binary Tree

Question Solution

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.

Runtime: **84 ms**

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool isBal;

bool isBalanced(TreeNode \*root) {

int depth = 0;

isBal = true;

postOrder(root, 0);

return isBal;

}

int postOrder(TreeNode \*p, int depth){

int maxD, minD;

depth ++;

if(!isBal) return depth;

if(p == NULL) return depth;

maxD = postOrder(p->left, depth);

minD = postOrder(p->right, depth);

if(minD>maxD){

int tmp = maxD;

maxD = minD;

minD = tmp;

}

if( maxD - minD > 1){

isBal = false;

}

return maxD;

}

};

Runtime: **288 ms**

class Solution {

public:

ListNode \*insertionSortList(ListNode \*head) {

ListNode \*virtual\_head = new ListNode(0);

ListNode \*p,\*pre,\*cur, \*next;

if(head == NULL || head->next == NULL) return head;

virtual\_head->next = head;

p = head->next;

next = p->next;

head->next = NULL;

while(p!=NULL){

pre = virtual\_head;

cur = virtual\_head->next;

while(cur!=NULL && cur->val <= p->val){

pre = pre->next;

cur = cur->next;

}

pre->next = p;

p->next = cur;

p = next;

if(p!=NULL) next = p->next;

}

return virtual\_head->next;

}

};

## Search a 2D Matrix

 Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

[

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

Given **target** = 3, return true.

**class** Solution {

**public**:

**bool** searchMatrix(vector<vector<**int**>>& matrix, **int** target) {

**int** n = matrix.size();

**if**(n==0) **return** **false**;

**int** m = matrix[0].size();

**if**(target<matrix[0][0] || target>matrix[n-1][m-1])

**return** **false**;

**int** mid, i, j;

**int** row;

i=0; j=n-1;

**while**(i<=j){

mid = (i+j)/2;

**int** val = matrix[mid][0];

**if**(val==target) **return** **true**;

**if**(mid+1>n-1 ||

(val<target && matrix[mid+1][0]>target)) {

row = mid;

**break**;

}

**if**(val<target) i = mid+1;

**else** j = mid-1;

}

i=0; j=m-1;

**while**(i<=j){

mid = (i+j)/2;

**int** val = matrix[row][mid];

**if**(val==target) **return** **true**;

**if**(val<target) i = mid + 1;

**if**(val>target) j = mid - 1;

}

**return** **false**;

}

};

## Set Matrix Zeroes

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.

**Follow up:**

Did you use extra space?  
A straight forward solution using O(*mn*) space is probably a bad idea.  
A simple improvement uses O(*m* + *n*) space, but still not the best solution.  
Could you devise a constant space solution?

**Solution: Use first row and first column to record 0s**

**class** Solution {

**public**:

**void** setZeroes(vector<vector<**int**>>& matrix) {

**bool** fr = **false**,fc = **false**;

**for**(**int** i = 0; i < matrix.size(); i++) {

**for**(**int** j = 0; j < matrix[0].size(); j++) {

**if**(matrix[i][j] == 0) {

**if**(i == 0) fr = **true**;

**if**(j == 0) fc = **true**;

matrix[0][j] = 0;

matrix[i][0] = 0;

}

}

}

**for**(**int** i = 1; i < matrix.size(); i++) {

**for**(**int** j = 1; j < matrix[0].size(); j++) {

**if**(matrix[i][0] == 0 || matrix[0][j] == 0)

matrix[i][j] = 0;

}

}

**if**(fr) {

**for**(**int** j = 0; j < matrix[0].size(); j++)

matrix[0][j] = 0;

}

**if**(fc) {

**for**(**int** i = 0; i < matrix.size(); i++)

matrix[i][0] = 0;

}

}

};

# Linked List

## ►Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

TreeNode\* sortedListToBST(ListNode\* head) {

if (head == NULL)

return NULL;

if (head->next == NULL)

return new TreeNode(head->val);

ListNode \*fast = head->next->next, \*slow = head;

while (fast != NULL && fast->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

TreeNode \*root = new TreeNode(slow->next->val);

root->right = sortedListToBST(slow->next->next);

slow->next = NULL;

root->left = sortedListToBST(head);

return root;

}

};

## Convert Sorted Array to Binary Search Tree

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

TreeNode\* sortedArrayToBST(vector<int>& nums) {

return helper(nums,0, nums.size()-1);

}

TreeNode\* helper(vector<int>& nums, int begin, int end){

if(end<begin) return NULL;

int mid = (end + begin)/2;

TreeNode \*root = new TreeNode(nums[mid]);

root->left = helper(nums,begin, mid-1);

root->right = helper(nums,mid+1, end);

return root;

}

};

# Binary Tree

## ►►Populating(填充) Next Right Pointers in Each Node

Given a binary tree

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Note:**

* You may only use **constant extra space**.
* You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,  
Given the following perfect binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

/\*\*

 \* Definition for binary tree with next pointer.

 \* struct TreeLinkNode {

 \* int val;

 \* TreeLinkNode \*left, \*right, \*next;

 \* TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}

 \* };

 \*/

**class** Solution {

**public**:

**void** connect(TreeLinkNode \*root) {

TreeLinkNode \*p,\*currentLevel;

**if**(root == NULL) **return** ;

root->next = NULL;

currentLevel = root;

p = root;

**while**(p!=NULL){

**if**(p->left!=NULL){

p->left->next = p->right;

}

**if**(p->right!=NULL && p->next!=NULL){

p->right->next = p->next->left;

}

p = p->next;

**if**(p==NULL){

p = currentLevel->left;

currentLevel = p;

}

}

}

};

## ►►►Populating Next Right Pointers in Each Node II

Follow up for problem "*Populating Next Right Pointers in Each Node*".

What if the given tree could be any binary tree? Would your previous solution still work?

**Note:**

* You may only use constant extra space.

For example,  
Given the following binary tree,

1

/ \

2 3

/ \ \

4 5 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ \

4-> 5 -> 7 -> NULL

/\*\*

 \* Definition for binary tree with next pointer.

 \* struct TreeLinkNode {

 \* int val;

 \* TreeLinkNode \*left, \*right, \*next;

 \* TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}

 \* };

 \*/

**class** Solution {

**public**:

**void** connect(TreeLinkNode \*root) {

TreeLinkNode \*now, \*tail, \*head;

now = root;

head = tail = NULL;

**while**(now){

**if** (now->left)

**if** (tail) tail = tail->next =now->left;

**else** head = tail = now->left;

**if** (now->right)

**if** (tail) tail = tail->next =now->right;

**else** head = tail = now->right;

now = now->next;

**if**(!now){

now = head;

head = tail=NULL;

}

}

}

};

## Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example,  
Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \* int val;

 \* TreeNode \*left;

 \* TreeNode \*right;

 \* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

**class** Solution {

**public**:

**void** flatten(TreeNode\* root) {

**if**(root == NULL ) **return**;

TreeNode \*left = root->left;

TreeNode \*right = root->right;

**if**(left!=NULL) flatten(left);

**if**(right!=NULL) flatten(right);

root->left = NULL;

**if**(left==NULL) root->right = right;

**else** {

root->right = left;

**if**(right!=NULL) {

TreeNode \*p=left;

**while**(p->right != NULL) p = p->right;

p->right = right;

}

}

}

};

## Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:  
You may assume that duplicates do not exist in the tree.

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

TreeNode\* buildTree(vector<int>& inorder, vector<int>& postorder) {

return buildTree(inorder, 0, inorder.size()-1, postorder, 0, postorder.size()-1);

}

TreeNode\* buildTree(vector<int>& inorder,int inbegin,int inend ,vector<int>& postorder, int postbegin,int postend) {

int n = inend - inbegin + 1;

if(n==0) return NULL;

int rootNum = postorder[postend];

int p;

for(p=inbegin; p!=inend&&inorder[p]!=rootNum; p++);

TreeNode \*root = new TreeNode(rootNum);

int left\_length = p - inbegin ;

int right\_length = inend - p;

root->left = buildTree(inorder,inbegin, p-1, postorder, postbegin, postbegin+left\_length-1);

root->right = buildTree(inorder,p+1, inend, postorder, postbegin+left\_length, postend-1);

return root;

}

};

## ►►Count Complete Tree Nodes

Given a **complete** binary tree, count the number of nodes. In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2hnodes inclusive at the last level h.

**172ms Solution, recursive, easy to understand:**

**class** Solution {

**public**:

**int** countNodes(TreeNode\* root) {

**if**(!root) **return** 0;

**int** hl=0, hr=0;

TreeNode \*l=root, \*r=root;

**while**(l) {hl++;l=l->left;}

**while**(r) {hr++;r=r->right;}

**if**(hl==hr) **return** (1<<hl)-1;

**return** 1+countNodes(root->left)+countNodes(root->right);

}

};

**80ms Solution, iterative:**

**class** Solution {

**public**:

**int** countNodes(TreeNode\* root) {

**if**(!root) **return** 0;

TreeNode\* p = root;

**int** height = 0;

**int** result = 0; //最后一行的叶子树

**while**(p->left) { height++; p = p->left; }

**long** power = 1<<height;

**int** depth = 0;

**while** (root->left) {

**if**(!root->right) **break**;

**else** {

depth++;

power /= 2;

p = root->right;

**int** count = depth;

**while**(p->left) {p=p->left;count++;}

**if** (count==height) { //左子树是满的

root = root->right;

result += power; //满二叉树最后一行叶子树是 2^(h-1)

} **else** {

root = root->left;

}

}

}

**return** result+(1<<height);

}

};

## ►►Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example:  
Given the below binary tree,

1

/ \

2 3

Return 6.

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \* int val;

 \* TreeNode \*left;

 \* TreeNode \*right;

 \* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

**class** Solution {

**public**:

**int** maxSum;

**int** maxPathSum(TreeNode\* root) {

**if**(root==NULL) **return** 0;

maxSum = INT\_MIN;

**int** localMaxSum;

maxPathSum(root, localMaxSum);

**return** maxSum;

}

**void** maxPathSum(TreeNode\* root, **int** &localMaxSum) {

localMaxSum = root->val;

**int** leftMax =INT\_MIN;

**int** rightMax = INT\_MIN;

**if**(root->left!=NULL) {

maxPathSum(root->left, leftMax);

localMaxSum = max(localMaxSum, root->val + leftMax);

}

**if**(root->right!=NULL) {

maxPathSum(root->right, rightMax);

localMaxSum = max(localMaxSum, root->val + rightMax);

}

maxSum = max(maxSum, localMaxSum);

**if**(root->left!=NULL && root->right!=NULL)

maxSum = max(maxSum, root->val + leftMax + rightMax);

}

};

# Binary Search Tree

## ►Binary Tree Inorder Traversal (Non Recursive)

vector<**int**> inorderTraversal(TreeNode\* root) {

stack<TreeNode\*> S;

vector<**int**> result;

**if**(root==NULL) **return** result;

TreeNode \*p = root;

**do**{

**while**(p!=NULL){

S.push(p);

p = p->left;

}

p = S.top();S.pop();

result.push\_back(p->val);

p = p->right;

}**while**(!S.empty() || p!=NULL);

**return** result;

}

## Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Test case:[10,5,15,#,#,6,20] false

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \* int val;

 \* TreeNode \*left;

 \* TreeNode \*right;

 \* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

**class** Solution {

**public**:

**bool** isValidBST(TreeNode\* root) {

**if**(root==NULL) **return** **true**;

**int** Min,Max;

**return** isValidBST(root, Min, Max);

}

**bool** isValidBST(TreeNode\* root, **int** &curMin, **int** &curMax) {

**if**(root==NULL) **return** **true**;

**int** Min = INT\_MAX, Max = INT\_MIN;

curMin = curMax = root->val;

**if**(root->left != NULL && (root->left->val >= root->val

|| !isValidBST(root->left, Min, Max)

|| Max >= root->val)) **return** **false**;

curMin = min(Min, root->val);

curMax = max(Max, root->val);

**if**(root->right!= NULL && (root->right->val <= root->val

|| !isValidBST(root->right,Min, Max)

|| Min <= root->val)) **return** **false**;

curMin = min(curMin, Min);

curMax = max(curMax, Max);

**return** **true**;

}

};

## How to handle duplicates in Binary Search Tree?

In a Binary Search Tree (BST), all keys in left subtree of a key must be smaller and all keys in right subtree must be greater. So a Binary Search Tree by definition has distinct keys.

How to allow duplicates where every insertion inserts one more key with a value and every deletion deletes one occurrence?

A Simple Solution is to allow same keys on right side (we could also choose left side). For example consider insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree

12

/ \

10 20

/ \ /

9 11 12

/ \

10 12

A **Better Solution** is to augment every tree node to store count together with regular fields like key, left and right pointers.

Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.

12(3)

/ \

10(2) 20(1)

/ \

9(1) 11(1)

Count of a key is shown in bracket

# Graph algorithms

# Breadth-First-Search(Shortest Path)

## ►Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

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

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

vector<vector<int>> result;

if(root==NULL) return result;

bool isAscend = true;

queue<TreeNode\*> q;

q.push(root); q.push(NULL);

vector<int> v;

while(!q.empty()){

TreeNode \*p = q.front();

q.pop();

if(p!=NULL){

v.push\_back(p->val);

if(p->left!=NULL) q.push(p->left);

if(p->right!=NULL) q.push(p->right);

}else if(p==NULL) {

if(q.size()!=0) q.push(NULL);

if(!isAscend) reverse(v.begin(), v.end());

result.push\_back(v);

v.clear();

isAscend = !isAscend;

}

}

return result;

}

};

## ►Binary Tree Right Side View(牢记：带level识别的宽度优先算法)

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example:Given the following binary tree,

1 <---

/ \

2 3 <---

\ \

5 4 <---

You should return [1, 3, 4].

Algorithm: Breadth-Fist Search. Use a “Sentinel value: NULL” to remember boundary of the levels.

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<int> rightSideView(TreeNode\* root) {

vector<int> result;

if(root==NULL) return result;

queue<TreeNode\*> q;

q.push(root); q.push(NULL);

TreeNode \*pre;

while(!q.empty()){

TreeNode \*p = q.front();

q.pop();

if(p!=NULL){

pre = p;

if(p->left!=NULL) q.push(p->left);

if(p->right!=NULL) q.push(p->right);

}else if(p==NULL) {

if(q.size()!=0) q.push(NULL);

result.push\_back(pre->val);

}

}

return result;

}

};

►Binary Tree Level Order Traversal II (牢记：带level识别的宽度优先算法)

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:

[

[15,7],

[9,20],

[3]

]

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<vector<int>> levelOrderBottom(TreeNode\* root) {

vector<vector<int>> result;

if(root==NULL) return result;

queue<TreeNode\*> q;

q.push(root); q.push(NULL);

vector<int> v;

while(!q.empty()){

TreeNode \*p = q.front();

q.pop();

if(p!=NULL){

v.push\_back(p->val);

if(p->left!=NULL) q.push(p->left);

if(p->right!=NULL) q.push(p->right);

}else if(p==NULL) {

if(q.size()!=0) q.push(NULL);

result.push\_back(v);

v.clear();

}

}

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

return result;

}

};

## ►Word Ladder

Given two words (beginWord and endWord), and a dictionary, find the length of shortest transformation sequence from beginWord to endWord, such that:

Only one letter can be changed at a time

Each intermediate word must exist in the dictionary

For example,

Given:  
start = "hit"  
end = "cog"  
dict = ["hot","dot","dog","lot","log"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",  
return its length 5.

Note:

Return 0 if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

class Solution {

public:

struct Word

{

string word;

int color;

int depth;

Word(string s,int c, int d){

word = s; color = c; depth = d;

}

};

int ladderLength(string beginWord, string endWord, unordered\_set<string>& wordDict) {

if(wordDict.count(beginWord)<=0) return 0;

queue<Word\*> q;

int N = beginWord.length();

Word \*w = new Word(beginWord,1,1);

q.push(w);

while(!q.empty()){

Word \*w = q.front();

q.pop();

if(w->color==1){

w->color = 0;

for(char c='a';c<='z';c++){

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

string s = w->word;

s[i] = c;

if(s.compare(endWord)==0 && wordDict.count(s)>0) return w->depth+1;

if(wordDict.count(s)>0 ){

Word\* tmp = new Word(s, 1, w->depth+1);

q.push(tmp);

wordDict.erase(s); //trick: delete from dict so that prevent duplicate access

}

}

}

}

}

return 0;

}

};

Improve from 300ms to 80ms : Tow-Way BFS

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-spring-2008/recitations/recitation12.pdf

# Depth-first Search

►**Course Schedule**

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, [[1,0],[0,1]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

**Solution: CLRS Page 614. Lemma 22.11:G is a DAG iff DFS yields no back edges. CLRS Page 611: G have back edge iff found gray vertex in DFS.**

class Solution {

public:

enum Color{White, Gray, Black};

bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites)

{

vector<int> empty\_v;

vector<vector<int> > graph(numCourses,empty\_v);

vector<Color> color(numCourses, White);

int i;

for(i = 0; i < prerequisites.size(); i++) {

pair<int, int> aPair = prerequisites[i];

int first = aPair.first;

int second = aPair.second;

graph[second].push\_back(first);

}

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

if(color[i] == White)

if(dfs(graph,color,i)==false) return false;

}

return true;

}

bool dfs(vector<vector<int>> &graph, vector<Color>& color, int i){

vector<int> &adj = graph[i];

color[i] = Gray;

for(auto j:adj){

cout<<j<<endl;

if(color[j]==Gray) return false;

if(color[j]==White) {

if(dfs(graph, color, j)==false) return false;

}

}

color[i] = Black;

return true;

}

};

## ►Number of Islands

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

11110  
11010  
11000  
00000

Answer: 1

Example 2:

11000  
11000  
00100  
00011

Answer: 3

class Solution {

public:

int numIslands(vector<vector<char>> &grid) {

int result = 0;

for(int i=0; i<grid.size(); i++){

for(int j=0; j<grid[i].size(); j++){

if(grid[i][j] == '1') {

result ++;

expend(grid, i, j);

}

}

}

return result;

}

void expend(vector<vector<char>> &grid, int i, int j){

grid[i][j] = '2';

if( i-1>=0 && grid[i-1][j]=='1') expend(grid, i-1, j);

if( i+1<grid.size() && grid[i+1][j]=='1') expend(grid, i+1, j);

if( j-1>=0 && grid[i][j-1]=='1') expend(grid, i, j-1);

if( j+1<grid[i].size() && grid[i][j+1]=='1') expend(grid, i, j+1);

}

};

## Surrounded Regions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

X X X X

X O O X

X X O X

X O X X

After running your function, the board should be:

X X X X

X X X X

X X X X

X O X X

class Solution {

public:

int n,m;

void solve(vector<vector<char>>& board) {

n = board.size();

if(n<=0) return;

m = board[0].size();

int i,j;

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

if(i==0 || i==n-1){

for(j=0;j<m;j++) dfs(board,i,j);

}else{

dfs(board,i,0);

dfs(board,i,m-1);

}

}

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

for(j=0;j<m;j++){

if(board[i][j]=='O') board[i][j]='X';

if(board[i][j]=='G') board[i][j]='O';

}

}

}

void dfs(vector<vector<char>>& board,int i, int j){

if(i<0 || i>=n || j<0 || j>=m) return;

if(board[i][j]=='O') {

board[i][j] = 'G';

// no need to consider the peripheral border, so the condition

// is i > 1, i < rows - 2, not i > 0, i < rows - 1.

// if use i > 0, i < rows - 1, DFS solution will get a Runtime Error, confusing!

if(i-1>0 && board[i-1][j]=='O') dfs(board, i-1, j);

if(i<n-2 && board[i+1][j]=='O') dfs(board, i+1, j);

if(j-1>0 && board[i][j-1]=='O') dfs(board, i, j-1);

if(j<m-2 && board[i][j+1]=='O') dfs(board, i, j+1);

}

}

};

## ►Course Schedule II ---- Toplogical-Sort

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1]

4, [[1,0],[2,0],[3,1],[3,2]]

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is[0,2,1,3].

Topological-Sort(G)

1. call DFS(G) to compute finishing times v.f for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

class Solution {

public:

int time;

enum Color{White, Gray, Black};

vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites) {

vector<int> result;

vector<int> empty\_v;

vector<vector<int> > graph(numCourses,empty\_v);

vector<Color> color(numCourses, White);

vector<int> finish\_time(numCourses, 0);

int i;

time = 0;

for(i = 0; i < prerequisites.size(); i++) {

pair<int, int> aPair = prerequisites[i];

int first = aPair.first;

int second = aPair.second;

graph[second].push\_back(first);

}

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

if(color[i] == White)

if(dfs(graph,color, finish\_time,result, i)==false) return empty\_v;

}

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

return result;

}

bool dfs(vector<vector<int>> &graph, vector<Color>& color,vector<int> &finish\_time,vector<int> &result, int i){

time ++;

vector<int> &adj = graph[i];

color[i] = Gray;

for(auto j:adj){

cout<<j<<endl;

if(color[j]==Gray) return false;

if(color[j]==White) {

if(dfs(graph, color, finish\_time, result, j)==false) return false;

}

}

color[i] = Black;

time ++;

finish\_time[i] = time;

result.push\_back(i);

return true;

}

};

# Back Tracking

## ►►N-Queens (经典回溯)

The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.



Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.

Each solution contains a distinct board configuration of the *n*-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

For example,  
There exist two distinct solutions to the 4-queens puzzle:

[

[".Q..", // Solution 1

"...Q",

"Q...",

"..Q."],

["..Q.", // Solution 2

"Q...",

"...Q",

".Q.."]

]

Question:

1. How to determing vertical, diagonal collision simplely? Niave solution: three vector<bool>

In this problem, we can go row by row, and in each position, we need to check if the column, the 45° diagonal and the 135° diagonal had a queen before.

**Solution A:** Directly check the validity of each position, 12ms:

**class** Solution {

**public**:

vector<vector<string>> solveNQueens(**int** n) {

vector<vector<string>> res;

vector<string> nQueens(n, string(n, '.'));

solveNQueens(res, nQueens, 0, n);

**return** res;

}

**private**:

**void** solveNQueens(vector<vector<string>> &res,

vector<string> &nQueens, **int** row, **int** &n)

{

**if** (row == n) {

res.push\_back(nQueens);

**return**;

}

**for** (**int** col = 0; col<n; ++col)

**if** (isValid(nQueens, row, col, n)) {

nQueens[row][col] = 'Q';

solveNQueens(res, nQueens, row + 1, n);

nQueens[row][col] = '.';

}

}

**bool** isValid(vector<string> &nQueens, **int** row,**int** col,**int** &n)

{

//check if the column had a queen before.

**for** (**int** i = 0; i != row; ++i)

**if** (nQueens[i][col] == 'Q')

**return** **false**;

//check if the 45° diagonal had a queen before.

**for** (**int** i=row-1, j=col-1; i>=0 && j>=0; --i,--j)

**if** (nQueens[i][j] == 'Q')

**return** **false**;

//check if the 135° diagonal` had a queen before.

**for** (**int** i=row-1, j=col+1; i>=0 && j<n; --i,++j)

**if** (nQueens[i][j] == 'Q')

**return** **false**;

**return** **true**;

}

};

**Solution B:** Use flag vectors as bitmask, 4ms:

The number of columns is n, the number of 45° diagonals is 2 \* n - 1, the number of 135° diagonals is also 2 \* n - 1. When reach [row, col], the column No. is col, the 45° diagonal No. is row + col and the 135° diagonal No. is n - 1 + col - row. We can use three arrays to mask if the column or the diagonal had a queen before, if not, we can put a queen in this position and continue.

**class** Solution {

**public**:

vector<vector<string> > solveNQueens(**int** n) {

vector<vector<string> > res;

vector<string> nQueens(n, string(n, '.'));

vector<**int**> flag\_col(n, 1);

vector<int> flag\_45(2 \* n - 1, 1);

vector<int> flag\_135(2 \* n - 1, 1);

solveNQueens(res, nQueens, flag\_col,flag\_45,flag\_135,0,n);

**return** res;

}

**private**:

**void** solveNQueens(vector<vector<string> > &res,

vector<string> &nQueens,

vector<**int**> &flag\_col,

vector<**int**> &flag\_45,

vector<**int**> &flag\_135,

**int** row, **int** &n)

{

**if** (row == n) {

res.push\_back(nQueens);

**return**;

}

**for** (**int** col = 0; col != n; ++col)

**if**(flag\_col[col]&&flag\_45[row+col]&&flag\_135[n-1+col-row]) {

flag\_col[col]=0;

flag\_45[row+col]=0;

flag\_135[n-1+col-row]=0;

nQueens[row][col] = 'Q';

solveNQueens(res,nQueens,flag\_col,flag\_45,flag\_135,row+1,n);

nQueens[row][col] = '.';

flag\_col[col]=1;

flag\_45[row+col]=1;

flag\_135[n-1+col-row]=1;

}

}

};

But we actually do not need to use three mask arrays, we just need one. Now, when reach[row, col], the subscript of column is col, the subscript of 45° diagonal is n + row + col and the subscript of 135° diagonal is n + 2 \* n - 1 + n - 1 + col - row.

**class** Solution {

**public**:

vector<vector<string>> solveNQueens(**int** n) {

vector<vector<string>> res;

vector<string> nQueens(n, string(n, '.'));

/\*

  flag[0] to flag[n-1] to mask if the column had a queen before.

  flag[n] to flag[3\*n-2] to mask if the 45° diagonal had a queen before.

  flag[3 \* n - 1] to flag[5 \* n - 3] to mask if the 135° diagonal had a queen before.

  \*/

vector<**int**> flag(5 \* n - 2, 1);

solveNQueens(res, nQueens, flag, 0, n);

**return** res;

}

**private**:

**void** solveNQueens(vector<vector<string>> &res,

vector<string> &nQueens,

vector<**int**> &flag,

**int** row, **int** &n)

{

**if** (row == n) {

res.push\_back(nQueens);

**return**;

}

**for** (**int** col = 0; col != n; ++col)

**if** (flag[col]&&flag[n+row+col]&&flag[4\*n-2+col-row]) {

flag[col] = 0;

flag[n + row + col] = 0;

flag[4 \* n - 2 + col - row] = 0;

nQueens[row][col] = 'Q';

solveNQueens(res, nQueens, flag, row + 1, n);

nQueens[row][col] = '.';

flag[col] = 1;

flag[n + row + col] = 1;

flag[4 \* n - 2 + col - row] = 1;

}

}

};

## ►►►N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.



**Solution A:** Directly check the validity of each position, 8ms:

**class** Solution {

**public**:

**int** totalNQueens(**int** n) {

**int** res = 0;

vector<string> nQueens(n, string(n, '.'));

solveNQueens(res, nQueens, 0, n);

**return** res;

}

**private**:

**void** solveNQueens(**int** &res, vector<string> &nQueens,

**int** row, **int** &n)

{

**if** (row == n) {

res ++;

**return**;

}

**for** (**int** col = 0; col<n; ++col)

**if** (isValid(nQueens, row, col, n)) {

nQueens[row][col] = 'Q';

solveNQueens(res, nQueens, row + 1, n);

nQueens[row][col] = '.';

}

}

**bool** isValid(vector<string> &nQueens, **int** row,**int** col,**int** &n)

{

//check if the column had a queen before.

**for** (**int** i = 0; i != row; ++i)

**if** (nQueens[i][col] == 'Q')

**return** **false**;

//check if the 45° diagonal had a queen before.

**for** (**int** i=row-1, j=col-1; i>=0 && j>=0; --i,--j)

**if** (nQueens[i][j] == 'Q')

**return** **false**;

//check if the 135° diagonal` had a queen before.

**for** (**int** i=row-1, j=col+1; i>=0 && j<n; --i,++j)

**if** (nQueens[i][j] == 'Q')

**return** **false**;

**return** **true**;

}

};

**Solution B:** Use flag vectors as bitmask, 0ms:

**class** Solution {

**public**:

**int** totalNQueens(**int** n) {

**int** res = 0;

vector<string> nQueens(n, string(n, '.'));

vector<**int**> flag\_col(n, 1);

vector<**int**> flag\_45(2 \* n - 1, 1);

vector<**int**> flag\_135(2 \* n - 1, 1);

solveNQueens(res, nQueens, flag\_col,flag\_45,flag\_135,0,n);

**return** res;

}

**private**:

**void** solveNQueens(**int** &res, vector<string> &nQueens,

vector<**int**> &flag\_col,

vector<**int**> &flag\_45,

vector<**int**> &flag\_135,

**int** row, **int** &n)

{

**if** (row == n) {

res++;

**return**;

}

**for** (**int** col = 0; col != n; ++col)

**if**(flag\_col[col]&&flag\_45[row+col]&&flag\_135[n-1+col-row]) {

flag\_col[col]=0;

flag\_45[row+col]=0;

flag\_135[n-1+col-row]=0;

nQueens[row][col] = 'Q';

solveNQueens(res,nQueens,flag\_col,flag\_45,flag\_135,row+1,n);

nQueens[row][col] = '.';

flag\_col[col]=1;

flag\_45[row+col]=1;

flag\_135[n-1+col-row]=1;

}

}

};

## ►Combination Sum (背：组合和满足target,元素可重复

Given a set of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***. The **same** repeated number may be chosen from ***C*** **unlimited** number of times.

**Note:**

* All numbers (including target) will be positive integers.
* Elements in a combination (*a*1, *a*2, … , *a*k) must be in non-descending order. (ie, *a*1 ≤ *a*2 ≤ … ≤ *a*k).
* The solution set must not contain duplicate combinations.

For example, given candidate set 2,3,6,7 and target 7,   
A solution set is:   
[7]   
[2, 2, 3]

**class** Solution {

**public**:

vector<vector<**int**> > combinationSum(vector<**int**> &candidates,

**int** target)

{

sort(candidates.begin(), candidates.end());

vector<vector<**int**> > res;

vector<**int**> combination;

combinationSum(candidates, target, res, combination, 0);

**return** res;

}

**private**:

**void** combinationSum(vector<**int**> &cand, **int** target,

vector<vector<**int**> > &res,

vector<**int**> &combi, **int** begin)

{

**if** (!target) {

res.push\_back(combi);

**return**;

}

**for** (**int** i=begin; i!=cand.size()&&target>=cand[i];++i){

combi.push\_back(cand[i]);

combinationSum(cand, target - cand[i], res, combi, i);

combi.pop\_back();

}

}

};

## ►Combination Sum II(背：组合和满足target，元素不可重复

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

Each number in ***C*** may only be used **once** in the combination.

**Note:**

* All numbers (including target) will be positive integers.
* Elements in a combination (*a*1, *a*2, … , *a*k) must be in non-descending order. (ie, *a*1 ≤ *a*2 ≤ … ≤ *a*k).
* The solution set must not contain duplicate combinations.

For example, given candidate set 10,1,2,7,6,1,5 and target 8,   
A solution set is:   
[1, 7]   
[1, 2, 5]   
[2, 6]   
[1, 1, 6]

Failed cases:

Input:**[1,1], 1**

Output:**[[1],[1]]**

Expected:**[[1]]**

Solution: The difference with Combination Sum is

1. Cannot use current number in sub problems

combinationSum2(cand, target-cand[i],res, combi, i+1);

1. skip duplicated subproblom

**while**(i<cand.size()-1 && cand[i]==cand[i+1]) i++;

**class** Solution {

**public**:

vector<vector<**int**>> combinationSum2(vector<**int**>& candidates,

**int** target)

{

sort(candidates.begin(), candidates.end());

vector<vector<**int**> > res;

vector<**int**> combination;

combinationSum2(candidates, target, res, combination, 0);

**return** res;

}

**private**:

**void** combinationSum2(vector<**int**> &cand, **int** target,

vector<vector<**int**> > &res,

vector<**int**> &combi, **int** begin)

{

**if**(target<0) **return**;

**if** (!target) {

res.push\_back(combi);

**return**;

}

**for** (**int** i=begin; i<cand.size();++i){

combi.push\_back(cand[i]);

combinationSum2(cand, target-cand[i],res, combi, i+1);

combi.pop\_back();

**while**(i<cand.size()-1 && cand[i]==cand[i+1]) i++;

}

}

};

## ► Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab",  
Return

[

["aa","b"],

["a","a","b"]

]

**class** Solution {

**public**:

vector<vector<string>> partition(string s) {

vector<vector<string> > ret;

**if**(s.empty()) **return** ret;

vector<string> path;

dfs(0, s, path, ret);

**return** ret;

}

**void** dfs(**int** index, string& s,

vector<string>& path,

vector<vector<string> >& ret)

{

**if**(index == s.size()) {

ret.push\_back(path);

**return**;

}

**for**(**int** i = index; i < s.size(); ++i) {

**if**(isPalindrome(s, index, i)) {

path.push\_back(s.substr(index, i - index + 1));

dfs(i+1, s, path, ret);

path.pop\_back();

}

}

}

**bool** isPalindrome(**const** string& s, **int** start, **int** end) {

**while**(start <= end) {

**if**(s[start++] != s[end--])

**return** **false**;

}

**return** **true**;

}

};

## Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:  
Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

**class** Solution {

**public**:

vector<string> restoreIpAddresses(string s) {

vector<string> ret;

**if**(s.empty()) **return** ret;

vector<string> path;

dfs(0, s, path, ret);

**return** ret;

}

**void** dfs(**int** index, string &s,

vector<string> &path, vector<string> &ret)

{

**int** i;

string substr;

**if**(path.size() > 4) **return**;

**if**(index == s.size() && path.size()==4){

string ss;

**for**(i=0;i<4;i++){

ss += path[i];

**if**(i!=3) ss += ".";

}

ret.push\_back(ss);

}

**for**(i=index; i<s.size() && i<index+3; ++i){

**if**(isValid(s, index, i)){

path.push\_back(s.substr(index, i-index+1));

dfs(i+1, s, path, ret);

path.pop\_back();

}

}

}

**bool** isValid(**const** string&s, **int** start, **int** end){

**int** n = end - start + 1;

**if**(n>3) **return** **false**;

**if**(n==1 && s[start]=='0') **return** **true**;

**if**(n>1 && s[start] =='0') **return** **false**;

**int** num = atoi(s.substr(start, n).c\_str());

**if**(num>0 && num<256) **return** **true**;

**return** **false**;

}

};

## Word Search in Maze

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,  
Given **board** =

[

["ABCE"],

["SFCS"],

["ADEE"]

]

**word** = "ABCCED", -> returns true,  
**word** = "SEE", -> returns true,  
**word** = "ABCB", -> returns false.

**class** Solution {

**public**:

**bool** exist(vector<vector<**char**>>& board, string word) {

**int** i,j;

**int** n = board.size();

**int** m = board[0].size();

**if**(word.length()==0) **return** **false**;

**for**(i=0;i<n;i++)

**for**(j=0;j<m;j++){

**if**(board[i][j]==word[0]){

**if**(exist(board,n,m,i,j,word,0)==**true**)

**return** **true**;

}

}

**return** **false**;

}

**bool** exist(vector<vector<**char**>>& board,**int** n,**int** m,

**int** i, **int** j, string &word, **int** k)

{

**if**(board[i][j]!=word[k]) **return** **false**;

**if**(k==word.length()-1) **return** **true**;

**char** c = board[i][j];

board[i][j] = 0;

**if**(i>0 && exist(board,n,m,i-1,j,word,k+1)==**true**) {

board[i][j] = c;

**return** **true**;

}

**if**(j>0 && exist(board,n,m,i,j-1,word,k+1)==**true**){

board[i][j] = c;

**return** **true**;

}

**if**(i<n-1 && exist(board,n,m,i+1,j,word,k+1)==**true**) {

board[i][j] = c;

**return** **true**;

}

**if**(j<m-1 && exist(board,n,m,i,j+1,word,k+1)==**true**){

board[i][j] = c;

**return** **true**;

}

board[i][j] = c;

**return** **false**;

}

};

## Combinations

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

For example,  
If *n* = 4 and *k* = 2, a solution is:

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

For example,  
If *n* = 4 and *k* = 3

f([1,4],3)={

1+f([2,4],2),

= {

2+f([3,4],1): path+3,ret.push; path+4,ret.push;

3+f([4,4],1): path+4,ret.push;

}

2+f([3,4],2),

= {

3+f([4,4],1): path+4,ret.push;

}

3+f([4,4],2) ,

= {

4-4+1 < 2: X

}

}

**class** Solution {

**public**:

vector<vector<**int**>> combine(**int** n, **int** k) {

vector<vector<**int**>> ret;

vector<**int**> path;

combineHelper(1,n,k,path,ret);

**return** ret;

}

**void** combineHelper(**int** s, **int** e, **int** k,

vector<**int**> &path,

vector<vector<**int**>> &ret)

{

**int** i;

**if**(e-s+1<k || k<1) **return**;

**if**(k==1){

**for**(i=s;i<=e;i++){

path.push\_back(i);

ret.push\_back(path);

path.pop\_back();

}

}

**for**(i=s;i<=e;i++){

path.push\_back(i);

combineHelper(i+1,e,k-1,path,ret);

path.pop\_back();

}

}

};

## ►►►Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.



A sudoku puzzle...



...and its solution numbers marked in red.

**Solution:**

**Tricky 1:** board[i][j]!='.' or not we need to process next position;

**Tricky 2:** change states before recursive call, change back after call

**Tricky 3:** use return bool value to purning

**bool** row[9][10];

**bool** column[9][10];

**bool** cube[3][3][10];

**bool** helper(**char**\*\* board, **int** i, **int** j){

**int** ii, jj;

**bool** ret;

**if**(i==9) **return** **true**;

jj = (j+1)%9;

ii = i + (j+1)/9;

**if**(board[i][j]!='.') **return** helper(board, ii, jj);

**for**(**int** k=0;k<9;k++){

**if**(!row[i][k] && !column[j][k] && !cube[i/3][j/3][k]){

board[i][j] = k+'1';

row[i][k] = **true**;

column[j][k] = **true**;

cube[i/3][j/3][k] = **true**;

ret = helper(board, ii, jj);

**if**(ret) **return** **true**;

board[i][j] = '.';

row[i][k] = **false**;

column[j][k] = **false**;

cube[i/3][j/3][k] = **false**;

}

}

}

**void** solveSudoku(**char**\*\* board, **int** boardRowSize, **int** boardColSize) {

**int** i,j,k;

memset(row, 0, **sizeof**(row));

memset(column, 0, **sizeof**(column));

memset(cube, 0, **sizeof**(cube));

**for**(i=0;i<9;i++){

**for**(j=0;j<9;j++){

**char** c = board[i][j];

**if**(c != '.'){

row[i][c-'1'] = **true**;

column[j][c-'1'] = **true**;

cube[i/3][j/3][c-'1'] = **true**;

}

}

}

helper(board, 0, 0);

}

# Dynamic Programming

http://people.cs.clemson.edu/~bcdean/dp\_practice/

## ►Unique Binary Search Trees

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?

For example,  
Given *n* = 3, there are a total of 5 unique BST's.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

Solution: The number of binary search trees visualized as  
  
Number of binary search trees =

(Number of ways root can be choosen)\*

(Number of Left binary search sub-trees) \*

(Number of Right binary search sub-trees)   
Now, since there are "n" nodes in BST and let, the number of BST be represented by C(n)*for n elements*.  
We can find the number of BSTs recursively as :

1. choose 1 as root, *no* element  on the left sub-tree. *n-1* elements on the right sub-tree.
2. Choose 2 as root, *1* element  on the left sub-tree. *n-2* elements on the right sub-tree.
3. Choose 3 as root, *2* element on the left sub-tree. *n-3* elements on the right sub-tree.

Similarly, for *i-th* element as the root, *i-1* elements on the left and *n-i* on the right. These sub-trees are also BSTs so we can write :  
***C(n) = C(0)C(n-1) + C(1)C(n-2) + .....+ C(i-1)C(n-i)..... + C(n-1)C(0)***  
C(0) = 1, as there is exactly 1 way to make a BST with 0 nodes. C(1) = 1, as there is exactly 1 way to make a BST with 1 node.  
   
The above summation dissolves into Catalan Number.

http://qph.is.quoracdn.net/main-qimg-4e0cdd4e5a4976015e051d38c2bac2ef?convert_to_webp=true

**class** Solution {

**public**:

**int** numTrees(**int** n) {

unordered\_map<**int**,**int**> map;

map[0] = 1;

map[1] = 1;

**return** numTrees(n, map);

}

**int** numTrees(**int** n, unordered\_map<**int**, **int**> &map){

**if**(map.count(n)) **return** map[n];

**int** sum = 0;

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

sum += numTrees(i-1, map) \* numTrees(n-i, map);

map[n] = sum;

**return** sum;

}

};

## ►House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

**Algorithm: use two optimal list, to record optimal solution at i if ith house is robbed and not robbed;**

class Solution {

public:

int rob(vector<int>& nums) {

int n = nums.size();

int i;

if(n<=0) return 0;

vector<int> R(n,0), notR(n,0);

R[0] = nums[0];

notR[0] = 0;

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

R[i] = notR[i-1] + nums[i];

notR[i] = max(R[i-1],notR[i-1]);

}

return max(R[n-1],notR[n-1]);

}

};

## 找小偷问题

有n个房间，其中一个房间有小偷。早上我们可以打开一个房间的门看小偷在不在里面，晚上小偷会向左边或者右边的房间走。现在给你一个开门的sequence，你输出这个sequence能不能保证找到小偷。比如：如果只有三个房间那么如果打开房间的sequence是{1，1}那么一定会找到小偷。因为如果小偷在中间那么第一天就会被找到，如果小偷在两边那么第二天一定回来到中间也会被找到。房间数为n，sequence长度为k跟着我开始brute force假设小偷在某个房间然后dfs所有路径，大概是O(n\*2^k)。 考官说好，如果考虑cut branch呢？跟着我就说可以拿一个n\*k的matrix跟着根据sequence来cut ranch，reduce到O(n\*2\*k)。他说有没有可能同时从所有房间开始呢？我说可以跟着直接在那个n\*kmatrix上做一个类似dp的东西。跟着reduce 到 O(n\*k)。他说有没有可能把space reduce呢？我说可以我只要O(n)的space跟着他就让我再写一个叫nextRow的function来实现O(n)space。

nextDay[k][n] : 第k天，第n个房间小偷是否可以survive  
nextDay[i][j] = (nextDay[i-1][j-1] or nextDay[i-1][j+1]) && strategy[i] != j  
  
    // O(n\*k) time, O(n) space  
    boolean canCatchTheft(int n, int strategy[]) {  
        int k = strategy.length;  
        // nextDay[i] means theft can survive in spot i or not on this day  
        boolean nextDay[] = new boolean[n];   
        boolean canSurvive, pre, a, b;  
        // init the first day  
        Arrays.fill(nextDay, true); nextDay[strategy[0]] = false;  
        for (int i = 1; i < k; ++i) {   
            canSurvive = false; pre = false;  
            for (int j = 0; j < n; ++j) {   
                a = j == 0 ? false : pre;  
                b = j == n - 1 ? false : nextDay[j + 1];  
                pre = nextDay[j]; // store current day for the next round  
                nextDay[j] = ((a || b) && strategy[i] != j) ? true : false;  
                if(nextDay[j] == true) canSurvive = true;   
            }  
            if (!canSurvive) return true;  
        }  
        return false;  
    }

**Solution 2: tow round , rob the beginner house**

class Solution {

public:

int robHelper(vector<int>& nums, int begin, int end){

if (begin == end) return nums[begin];

vector<int> money(end - begin + 1, 0);

money[0] = nums[begin];

money[1] = max(nums[begin], nums[begin + 1]);

for (int i = 2; i <= end - begin; i++)

money[i] = max(money[i - 1], money[i - 2] + nums[begin + i]);

return money[money.size() - 1];

}

int rob(vector<int>& nums) {

if (nums.empty()) return 0;

int n = nums.size();

if (n == 1) return nums[0];

return max(robHelper(nums, 0, n - 1), robHelper(nums, 1, n - 1));

}

};

## ►House Robber II

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Solution: This problem is a little tricky at first glance. However, if you have finished the House Robber problem, this problem can simply be decomposed into two House Robber problems. Suppose there are n houses, since house 0 and n - 1 are now neighbors, we cannot rob them together and thus the solution is now the maximum of

Rob houses 0 to n - 2;

Rob houses 1 to n - 1.

The code is as follows (some edge cases are handled explicitly).

class Solution {

public:

int robHelper(vector<int>& nums, int begin, int end){

if (begin == end) return nums[begin];

vector<int> money(end - begin + 1, 0);

money[0] = nums[begin];

money[1] = max(nums[begin], nums[begin + 1]);

for (int i = 2; i <= end - begin; i++)

money[i] = max(money[i - 1], money[i - 2] + nums[begin + i]);

return money[money.size() - 1];

}

int rob(vector<int>& nums) {

if (nums.empty()) return 0;

int n = nums.size();

if (n == 1) return nums[0];

return max(robHelper(nums, 0, n - 2), robHelper(nums, 1, n - 1));

}

};

## ►Dungeon Game(地牢)

The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

**Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.**

For example, given the dungeon below, the initial health of the knight must be at least **7** if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

|  |  |  |
| --- | --- | --- |
| -2 (K) | -3 | 3 |
| -5 | -10 | 1 |
| 10 | 30 | -5 (P) |

**Notes:**

* The knight's health has no upper bound.
* Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

class Solution {

public:

int calculateMinimumHP(vector<vector<int> > &dungeon) {

int i,j;

int m = dungeon.size();

int n = dungeon[m-1].size();

vector<int> minEver( dungeon[m-1].size(), 0 );

minEver[n-1] = 1-dungeon[m-1][n-1];

for(j=n-2;j>=0;j--) {

if(minEver[j+1]<=0) minEver[j] = 1 - dungeon[m-1][j];

else minEver[j] = minEver[j+1] - dungeon[m-1][j];

}

for(i=m-2;i>=0;i--){

if(minEver[n-1]<=0) minEver[n-1] = 1 - dungeon[i][n-1];

else minEver[n-1] -= dungeon[i][n-1];

for(j=n-2;j>=0;j--){

if (minEver[j]<minEver[j+1]){

if(minEver[j]<=0) minEver[j] = 1-dungeon[i][j];

else minEver[j] = minEver[j]-dungeon[i][j];

}else{

if(minEver[j+1]<=0) minEver[j] = 1-dungeon[i][j];

else minEver[j] = minEver[j+1]-dungeon[i][j];

}

}

}

if(minEver[0] <= 0 ) return 1;

else return minEver[0];

}

};

## ►Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,  
Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

**Solution and Precautions: Dynamic Programming**

**This could be solved by DP. Let F(n) denote the number of decode ways for a string of length n. The following is the recursive formula:**

**F(n) = F(n-1) + F(n-2)**

**if s[n] is a valid encoding digit and s[n-1]s[n] is also a valid encoding**

**F(n) = F(n-1)**

**if s[n] is a valid encoding digit and s[n-1]s[n] is NOT a valid encoding**

**F(n) = F(n-2)**

**if s[n] is NOT a valid encoding digit and s[n-1]s[n] is  a valid encoding.**

**F(n) = 0**

**if s[n] is NOT a valid encoding digit and s[n-1]s[n] is NOT  a valid encoding**

**int** numDecodings(string s) {

**int** n = s.size();

**if**(n == 0 || s[0] == '0') **return** 0;

**if**(n == 1) **return** 1;

**int** res = 0,fn\_1 = 1,fn\_2 = 1;

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

**int** temp = fn\_1;

**if**(isValid(s[i])&&isValid(s[i-1],s[i])) fn\_1 = fn\_1+fn\_2;

**if**(!isValid(s[i])&&isValid(s[i-1],s[i])) fn\_1 = fn\_2;

**if**(isValid(s[i])&&!isValid(s[i-1],s[i])) fn\_1 = fn\_1;

**if**(!isValid(s[i])&&!isValid(s[i-1],s[i])) **return** 0;

fn\_2 = temp;

}

**return** fn\_1;

}

**bool** isValid(**char** a,**char** b){

**return** a == '1'||(a == '2' && b <='6');

}

**bool** isValid(**char** a){

**return** a != '0';

}

## ►Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

Hint: “Hash Table”, “Two Pointers”

Solution: The basic idea is, keep a hashmap which stores the characters in string as keys and their positions as values, and keep two pointers which define the max substring. move the right pointer to scan through the string , and meanwhile update the hashmap. If the character is already in the hashmap, then move the left pointer to the left of the same character last found. Note that the two pointers can only move forward.

class Solution {

public:

int lengthOfLongestSubstring(string s) {

vector<int> flag(256, -1);

int start = 0, result = 0;

for (int i = 0; i != s.size(); ++i) {

if (flag[s[i]] >= start)

start = flag[s[i]] + 1;

flag[s[i]] = i;

result = max(result, i - start + 1);

}

return result;

}

};

## Unique Paths

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



Above is a 3 x 7 grid. How many possible unique paths are there?

**Note:** *m* and *n* will be at most 100.

**Solution:** This is a fundamental DP problem. First of all, let's make some observations.

Since the robot can only move right and down, when it arrives at a point, there are only two possibilities:

1. It arrives at that point from above (moving down to that point);

2. It arrives at that point from left (moving right to that point).

Thus, we have the following state equations: suppose the number of paths to arrive at a point(i, j) is denoted as P[i][j], it is easily concluded that P[i][j] = P[i - 1][j] + P[i][j - 1].

The boundary conditions of the above equation occur at the leftmost column (P[i][j - 1] does not exist) and the uppermost row (P[i - 1][j] does not exist). These conditions can be handled by initialization (pre-processing) --- initialize P[0][j] = 1, P[i][0] = 1 for all valid i, j. Note the initial value is 1 instead of 0!

Now we can write down the following (unoptimized) code.

**int** uniquePaths(**int** m, **int** n) {

**vector**<**vector**<**int**> > path(m, **vector**<**int**> (n, 1));

**for** (**int** i = 1; i < m; i++)

**for** (**int** j = 1; j < n; j++)

path[i][j] = path[i - 1][j] + path[i][j - 1];

**return** path[m - 1][n - 1];

}

As can be seen, the above solution runs in O(n^2) time and costs O(n^2) space. However, you may have observed that each time when we update path[i][j], we only need path[i - 1][j](at the same column) and path[i][j - 1] (at the left column). So it is enough to maintain two columns (the current column and the left column) instead of maintaining the full m\*n matrix. Now the code can be optimized to have O(n) space complexity.

**int** uniquePaths(**int** m, **int** n) {

**vector**<**int**> pre(m, 1);

**vector**<**int**> cur(m, 1);

**for** (**int** j = 1; j < n; j++) {

**for** (**int** i = 1; i < m; i++)

cur[i] = cur[i - 1] + pre[i];

swap(pre, cur);

}

**return** pre[m - 1];

}

Further inspecting the above code, we find that keeping two columns is used to recover pre[i], which is just cur[i] before its updates. So there is even no need to use two vectors and one is just enough. Now the space is further saved and the code also gets much shorter.

**int** uniquePaths(**int** m, **int** n) {

**vector**<**int**> cur(m, 1);

**for** (**int** j = 1; j < n; j++)

**for** (**int** i = 1; i < m; i++)

cur[i] += cur[i - 1];

**return** cur[m - 1];

}

Well, till now, I guess you may even want to optimize it to O(1) space complexity since the above code seems to rely on only cur[i] and cur[i - 1]. You may think 2 variables is enough? Well, I am not quite sure about this point. However, since cur needs to be updated totally for n - 1 times, it means that all of its values need to saved for next full update and I guess two variables is not enough. If you are able to do it in O(1) space, please tell me about it.

## Unique Paths II

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

[

[0,0,0],

[0,1,0],

[0,0,0]

]

The total number of unique paths is 2.

**Note:** *m* and *n* will be at most 100.

Failed Cases:

Input:**[[1]]**

Output:**1**

Expected:**0**

Input:**[[1,0]]**

Output:**1**

Expected:**0**

Input:**[[0],[1]]**

Output:**1**

Expected:**0**

**class** Solution {

**public**:

**int** uniquePathsWithObstacles(vector<vector<**int**>>& obstacleGrid) {

**int** n = obstacleGrid.size();

**if**(n==0) **return** 0;

**int** m = obstacleGrid[0].size();

**int** i,j;

vector<**int**> cur(m, 1);

**int** flag = 1;

**for** (i = 0; i < m; i++){

**if**(obstacleGrid[0][i]==1) flag = 0;

cur[i] = flag;

}

flag = 1;

**for** (j = 1; j < n; j++){

**if**(obstacleGrid[j][0]==1 || cur[0]==0) flag = 0;

cur[0] = flag;

**for** (i = 1; i < m; i++){

**if**(obstacleGrid[j][i]==1) cur[i]=0;

**else** **if**(obstacleGrid[j][i-1]!=1) cur[i] += cur[i - 1];

}

}

**return** cur[m - 1];

}

};

**class** Solution {

## ►►Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4],  
the contiguous subarray [4,−1,2,1] has the largest sum = 6.

**More practice:**

If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution: Base case: 1 element, return nums[0]

Other cases:

* If dp[i-1] < 0, dp[i] = nums[i]
* if dp[i-1] >0, dp[i] = nums[i] + dp[i-1]

then pick the max sum.

We only need dp[i-1], so i use prev to record it, the space complexity is reduced to O(1).

Key is find the negative edges.

**public**:

**int** maxSubArray(vector<**int**>& nums) {

**int** n = nums.size();

**if**(n==0) **return** 0;

**int** pre = nums[0];

**int** cur = nums[0];

**int** max\_ret = nums[0];

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

**if**(pre>0) cur = pre + nums[i];

**else** cur = nums[i];

pre = cur;

max\_ret = max(max\_ret, cur);

}

**return** max\_ret;

}

};

## Maximal Square

Question Solution

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

1 0 1 0 0

1 0 1 1 1

1 1 1 1 1

1 0 0 1 0

Return 4.

Solution: It's actually to keep recording the max n\*n window at each cell of the matrix. At each cell, we define that the dynamic programming status at that cell is - if I am the most left-top guy of a square, how big the square I can build. With this definition, this status will be transferrable to the guys, right, below, and right below me.

16ms:

**class** Solution {

**public**:

**int** maximalSquare(vector<vector<**char**>>& matrix) {

**int** n = matrix.size();

**if**(n==0) **return** 0;

**int** m = matrix[0].size();

**int** res = 0;

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

**for**(**int** j=0;j<m;j++){

**int** tmp = maximalSquare(matrix,n,m,i,j);

res = max(tmp, res);

}

**return** res;

}

**int** maximalSquare(vector<vector<**char**>>& matrix,

**int** n, **int** m, **int** i,**int** j)

{

**int** k;

**for**(k=0;i+k<n&&j+k<m;k++){

**for**(**int** col=0;col<=k;col++)

**if**(matrix[i+k][j+col]=='0') **return** k\*k;

**for**(**int** row=0;row<k;row++)

**if**(matrix[i+row][j+k]=='0') **return** k\*k;

}

**return** k\*k;

}

};

## ►►►Edit Distance

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:

a) Insert a character  
b) Delete a character  
c) Replace a character

This is a classic problem of Dynamic Programming. We define the state dp[i][j] to be the minimum number of operations to convert word1[0..i - 1] to word2[0..j - 1]. The state equations have two cases: the boundary case and the general case. Note that in the above notations, both i and j take values starting from 1.

For the boundary case, that is, to convert a string to an empty string, it is easy to see that the mininum number of operations to convert word1[0..i - 1] to "" requires at least i operations (deletions). In fact, the boundary case is simply:

1. dp[i][0] = i;
2. dp[0][j] = j.

Now let's move on to the general case, that is, convert a non-empty word1[0..i - 1] to another non-empty word2[0..j - 1]. Well, let's try to break this problem down into smaller problems (sub-problems). Suppose we have already known how to convert word1[0..i - 2] toword2[0..j - 2], which is dp[i - 1][j - 1]. Now let's consider word[i - 1] and word2[j - 1]. If they are euqal, then no more operation is needed and dp[i][j] = dp[i - 1][j - 1]. Well, what if they are not equal?

If they are not equal, we need to consider three cases:

1. Replace word1[i - 1] by word2[j - 1] (dp[i][j] = dp[i - 1][j - 1] + 1 (for replacement));
2. Delete word1[i - 1] and word1[0..i - 2] = word2[0..j - 1] (dp[i][j] = dp[i - 1][j] + 1 (for deletion));
3. Insert word2[j - 1] to word1[0..i - 1] and word1[0..i - 1] + word2[j - 1] = word2[0..j - 1] (dp[i][j] = dp[i][j - 1] + 1 (for insertion)).

Make sure you understand the subtle differences between the equations for deletion and insertion. For deletion, we are actually converting word1[0..i - 2] to word2[0..j - 1], which costs dp[i - 1][j], and then deleting the word1[i - 1], which costs 1. The case is similar for insertion.

Putting these together, we now have:

1. dp[i][0] = i;
2. dp[0][j] = j;
3. dp[i][j] = dp[i - 1][j - 1], if word1[i - 1] = word2[j - 1];
4. dp[i][j] = min(dp[i - 1][j - 1] + 1, dp[i - 1][j] + 1, dp[i][j - 1] + 1), otherwise.

The above state equations can be turned into the following code directly.

**int** minDistance(string word1, string word2) {

**int** m = word1.length(), n = word2.length();

vector<vector<**int**> > dp(m + 1, vector<**int**> (n + 1, 0));

**for** (**int** i = 1; i <= m; i++)

dp[i][0] = i;

**for** (**int** j = 1; j <= n; j++)

dp[0][j] = j;

**for** (**int** i = 1; i <= m; i++) {

**for** (**int** j = 1; j <= n; j++) {

**if** (word1[i - 1] == word2[j - 1])

dp[i][j] = dp[i - 1][j - 1];

**else** dp[i][j] = min(dp[i-1][j-1],

min(dp[i][j-1], dp[i-1][j]))+1;

}

}

**return** dp[m][n];

}

Well, you may have noticed that each time when we update dp[i][j], we only need dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]. In fact, we need not maintain the full m\*n matrix. Instead, maintaing one column is enough. The code can be optimized to O(m) or O(n) space, depending on whether you maintain a row or a column of the original matrix.

The optimized code is as follows.

**int** minDistance(string word1, string word2) {

**int** m = word1.length(), n = word2.length();

vector<**int**> cur(m + 1, 0);

**for** (**int** i = 1; i <= m; i++)

cur[i] = i;

**for** (**int** j = 1; j <= n; j++) {

**int** pre = cur[0];

cur[0] = j;

**for** (**int** i = 1; i <= m; i++) {

**int** temp = cur[i];

**if** (word1[i - 1] == word2[j - 1])

cur[i] = pre;

**else** cur[i] = min(pre + 1, min(cur[i]+1,cur[i-1]+1));

pre = temp;

}

}

**return** cur[m];

}

Well, if you find the above code hard to understand, you may first try to write a two-column version that explicitly maintains two columns (the previous column and the current column) and then simplify the two-column version into the one-column version like the above code :-)

## ►►►Scramble String

Given a string *s1*, 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

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 *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.

Solution 1: 4ms

**bool** isScramble(string s1, string s2) {

**if**(s1==s2)

**return** **true**;

**int** len = s1.length();

**int** count[26] = {0};

**for**(**int** i=0; i<len; i++) {

count[s1[i]-'a']++;

count[s2[i]-'a']--;

}

**for**(**int** i=0; i<26; i++) {

**if**(count[i]!=0)

**return** **false**;

}

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

**if**( isScramble(s1.substr(0,i), s2.substr(0,i)) &&

isScramble(s1.substr(i), s2.substr(i)))

**return** **true**;

**if**( isScramble(s1.substr(0,i), s2.substr(len-i)) &&

isScramble(s1.substr(i), s2.substr(0,len-i)))

**return** **true**;

}

**return** **false**;

}

Solution 2: but 52ms

A more concise C++ DP version. dp[i][j][l] means whether s2.substr(j,l) is a scrambled string of s1.substr(i,l) or not.

**bool** isScramble(string s1, string s2) {

**int** len=s1.size();

**bool** dp[100][100][100]={**false**};

**for** (**int** i=len-1;i>=0;i--)

**for** (**int** j=len-1;j>=0;j--) {

dp[i][j][1]=(s1[i]==s2[j]);

**for** (**int** l=2;i+l<=len && j+l<=len;l++) {

**for** (**int** n=1;n<l;n++) {

dp[i][j][l]|=dp[i][j][n]&&dp[i+n][j+n][l-n];

dp[i][j][l]|=dp[i][j+l-n][n]&&dp[i+n][j][l-n];

}

}

}

**return** dp[0][0][len];

}

## ►►►Interleaving String 交错

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.

For example,  
Given:  
*s1* = "aabcc",  
*s2* = "dbbca",

When *s3* = "aadbbcbcac", return true.  
When *s3* = "aadbbbaccc", return false.

**bool** isInterleave(string s1, string s2, string s3) {

**int** n = s1.length();

**int** m = s2.length();

**if**(s3.length() != n + m) **return** **false**;

**int** i,j;

**bool** dp[n+1][m+1];

dp[0][0] = **true**;

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

dp[i][0] = dp[i-1][0]&&(s1[i-1]==s3[i-1]);

**for**(j=1;j<=m;j++)

dp[0][j] = dp[0][j-1]&&(s2[j-1]==s3[j-1]);

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

**for**(j=1;j<=m;j++){

dp[i][j] = dp[i-1][j]&&(s1[i-1]==s3[i+j-1]) ||

dp[i][j-1]&&(s2[j-1]==s3[i+j-1]);

}

}

**return** dp[n][m];

}

## ►►►Distinct Subsequences

Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example:  
**S** = "rabbbit", **T** = "rabbit"

Return 3.

/\*\*

 \* Solution (DP):

 \* We keep a m\*n matrix and scanning through string S, while

 \* m = T.length() + 1 and n = S.length() + 1

 \* and each cell in matrix Path[i][j] means the number of

\* distinct subsequences of T.substr(1...i) in S(1...j)

 \*

 \* Path[i][j] = Path[i][j-1] (discard S[j])

 \* + Path[i-1][j-1] (S[j]==T[i] and we are going to use S[j])

 \* or 0 (S[j] != T[i] so we could not use S[j])

 \* while Path[0][j] = 1 and Path[i][0] = 0.

 \*/

**int** numDistinct(string S, string T) {

**int** m = T.length();

**int** n = S.length();

**if** (m > n) **return** 0; // impossible for subsequence

vector<vector<**int**>> path(m+1, vector<**int**>(n+1, 0));

**for** (**int** k = 0; k <= n; k++) path[0][k] = 1; // init

**for** (**int** i = 1; i <= m; i++) {

**for** (**int** j = 1; j <= n; j++) {

path[i][j] = path[i][j-1] +

(T[i-1]==S[j-1] ? path[i-1][j-1] : 0);

}

}

**return** path[m][n];

}

/\*\*

 \* Further optimization could be made that we can use only 1D

\* array instead of a matrix, since we only need data from

\* last time step.

 \*/

**int** numDistinct(string S, string T) {

**int** m = T.length();

**int** n = S.length();

**if** (m > n) **return** 0; // impossible for subsequence

vector<**int**> path(m+1, 0);

path[0] = 1; // initial condition

**for** (**int** j = 1; j <= n; j++) {

// traversing backwards so we are

// using path[i-1] from last time step

**for** (**int** i = m; i >= 1; i--) {

path[i] = path[i] +

(T[i-1] == S[j-1] ? path[i-1] : 0);

}

}

**return** path[m];

}

## Minimum Path Sum

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.

**class** Solution {

**public**:

**int** minPathSum(vector<vector<**int**>>& a) {

**int** m=a.size();

**if**(m==0) **return** 0;

**int** n= a[0].size();

**for**(**int** i = 0 ; i<m; i++ ){

**for**(**int** j=0; j<n ; j++){

**int** left= (j==0) ? INT\_MAX : a[i][j-1];

**int** up = (i==0) ? INT\_MAX : a[i-1][j];

**if**(i==0 && j==0) **continue**;

a[i][j] += min(left, up );

}

}

**return** a[m-1][n-1];

}

};

# Greedy

## ►►Jump Game II

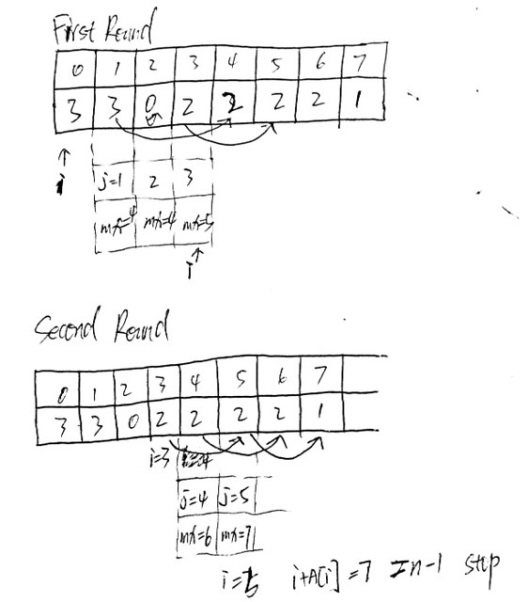
Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:  
Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)



**class** Solution {

**public**:

**int** jump(vector<**int**> &A) {

**int** i = 0, j = 1, cnt = 0, mx;

**int** n = A.size();

**if** (n == 1) **return** 0;

**while** (i < n - 1 && i + A[i] < n - 1) {

**for** (mx = j; j <= i + A[i]; j++) {

mx = (mx + A[mx] <= j + A[j]) ? j : mx;

}

i = mx; cnt++;

}

**return** ++cnt; /\* One more step to last index. \*/

}

};

## ►Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:  
The solution is guaranteed to be unique.

class Solution {

public:

int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {

int i;

for( i= 0; i < gas.size(); i++) {

gas[i] -= cost[i];

}

int sum = 0;

int result = 0;

int n = gas.size();

for(i = 0; i < n \* 2 - 1; i++) {

sum += gas[i % n];

if(sum < 0) {

result = i + 1;

if(result >= n) {

return -1;

}

sum = 0;

}

}

return result;

}

};

# Puzzles & Math

## Subsets II

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.

**Note:**

Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

For example,  
If ***nums*** = [1,2,2], a solution is:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

If ***nums*** = [1,2,2,2], a solution is:

[

[],

[1],

[2], ----num=2 pre\_count = 1

[1,2],

[2], X ----num=2 pre\_count = 2

[1,2],X

[2,2],

[1,2,2], ----num=2 pre\_count = 2

[2], X

[1,2], X

[2,2], X

[1,2,2],X

[2,2], X

[1,2,2],X

[2,2,2],

[1,2,2,2], ----num=2 pre\_count = 2

]

**class** Solution {

**public**:

vector<vector<**int**>> subsetsWithDup(vector<**int**>& nums) {

vector<vector<**int**>> ret;

vector<**int**> subset;

**int** n = nums.size();

**if**(n==0) **return** ret;

sort(nums.begin(), nums.end());

ret.push\_back(subset);

subset.push\_back(nums[0]);

ret.push\_back(subset);

**int** count, pre\_count = 1;

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

count = 0;

**int** num = nums[i];

**int** cur\_size = ret.size();

**int** j=0;

**if**(num == nums[i-1]) j = cur\_size - pre\_count;

**for**(;j<cur\_size;j++){

subset = ret[j];

subset.push\_back(num);

ret.push\_back(subset);

count++;

}

pre\_count = count;

}

**return** ret;

}

};

## ►Divide Two Integers

In this problem, we are asked to divide two integers. However, we are not allowed to use division, multiplication and mod operations. So, what else can we use? Yeah, bit manipulations.

Let's do an example and see how bit manipulations work.

Suppose we want to divide 15 by 3, so 15 is dividend and 3 is divisor. Well, division simply requires us to find how many times we can subtract the divisor from the the dividendwithout making the dividend negative.

Let's get started. We subtract 3 from 15 and we get 12, which is positive. Let's try to subtract more. Well, we **shift** 3 to the left by 1 bit and we get 6. Subtracting 6 from 15 still gives a positive result. Well, we shift again and get 12. We subtract 12 from 15 and it is still positive. We shift again, obtaining 24 and we know we can at most subtract 12. Well, since 12 is obtained by shifting 3 to left twice, we know it is 4 times of 3. How do we obtain this 4? Well, we start from 1 and shift it to left twice at the same time. We add 4 to an answer (initialized to be 0). In fact, the above process is like 15 = 3 \* 4 + 3. We now get part of the quotient (4), with a remainder 3.

Then we repeat the above process again. We subtract divisor = 3 from the remainingdividend = 3 and obtain 0. We know we are done. No shift happens, so we simply add 1 << 0to the answer.

Now we have the full algorithm to perform division.

According to the problem statement, we need to handle some exceptions, such as overflow.

Well, two cases may cause overflow:

1. divisor = 0;
2. dividend = INT\_MIN and divisor = -1 (because abs(INT\_MIN) = INT\_MAX + 1).

Of course, we also need to take the sign into considerations, which is relatively easy.

Putting all these together, we have the following code.

**class** Solution {

**public**:

**int** divide(**int** dividend, **int** divisor) {

**if** (!divisor || (dividend == INT\_MIN && divisor == -1))

**return** INT\_MAX;

**int** sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;

**long** **long** dvd = labs(dividend);

**long** **long** dvs = labs(divisor);

**int** res = 0;

**while** (dvd >= dvs) {

**long** **long** temp = dvs, multiple = 1;

**while** (dvd >= (temp << 1)) {

temp <<= 1;

multiple <<= 1;

}

dvd -= temp;

res += multiple;

}

**return** res \* sign;

}

};

## Rotate Array

Rotate an array of n elements to the right by k steps.

For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

Could you do it in-place with O(1) extra space?

Algorithm 1: Reverse two times.

class Solution {

public:

void rotate(int nums[], int n, int k) {

int i,j,tmp;

k = k%n;

for(i=0;i<n/2;i++){

j = n-1-i;

swap( nums[i], nums[j]);

}

for(i=0;i<k/2;i++){

j = k-1-i;

swap( nums[i], nums[j]);

}

for(i=k;i<(n+k)/2;i++){

j = n-1-i+k;

swap( nums[i], nums[j]);

}

}

void swap(int& x, int& y){

int tmp = x;

x = y;

y = tmp;

}

};

## ►Fraction to Recurring Decimal

 Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

Given numerator = 1, denominator = 2, return "0.5".

Given numerator = 2, denominator = 1, return "2".

Given numerator = 2, denominator = 3, return "0.(6)".

Given numerator = 1, denominator = 19, return "0.(052631578947368421)".

class Solution {

public:

string fractionToDecimal(int64\_t n, int64\_t d) {

// zero numerator

if (n == 0) return "0";

string res;

// determine the sign

if (n < 0 ^ d < 0) res += '-';

// remove sign of operands

n = abs(n), d = abs(d);

// append integral part

res += to\_string(n / d);

// in case no fractional part

if (n % d == 0) return res;

res += '.';

unordered\_map<int, int> map;

// simulate the division process

for (int64\_t r = n % d; r; r %= d) {

// meet a known remainder

// so we reach the end of the repeating part

if (map.count(r) > 0) {

res.insert(map[r], 1, '(');

res += ')';

break;

}

// the remainder is first seen

// remember the current position for it

map[r] = res.size();

r \*= 10;

// append the quotient digit

res += to\_string(r / d);

}

return res;

}

};

## Factorial Trailing Zeroes

Given an integer n, return the number of trailing zeroes in n!.

Note: Your solution should be in logarithmic time complexity.

class Solution {

public:

int trailingZeroes(int n) {

int res=0;

while(n){

n/=5;

res+=n;

}

return res;

}

};

## ►Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Solution:It is easy to derive a solution based on bucket sort algorithm. we only need to store the min&max value in each bucket, and calculate the gap between non-empty buckets, you can also store them in a min and max array separately.

Normalize every node.

class Solution {

public:

int maximumGap(vector<int> &num) {

int mi, ma;

int n = num.size();

if(n < 2) return 0;

mi = ma = num[0];

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

mi = min(mi, num[i]);

ma = max(ma, num[i]);

}

if(mi == ma) return 0;

vector<int> bs(2\*n, -1); //store mins and maxs for each bucket

double gap = (ma - mi) \* 1.0 / (n - 1); // compute average length of each bucket

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

int nom = (int)((num[i] - mi) / gap); // compute the bucket index for a value

int ind = 2 \* nom; //index

if(bs[ind] == -1) bs[ind] = num[i];

else bs[ind] = min(bs[ind], num[i]);

if(bs[ind+1] == -1) bs[ind+1] = num[i];

else bs[ind+1] = max(bs[ind+1], num[i]);

}

int ans = 0;

int i = 0;

int m = bs.size();

while(i < m) {

while(i < m && bs[i] == -1) i++;

int j = 1;

while((i+j) < m && bs[i+j] == -1) j++;

if((i+j) < m && bs[i+j] >= 0) ans = max(ans, bs[i+j]-bs[i]);

i = i+j;

} // calculate gaps, I believe you can do it more elegantly~

return ans;

}

};

## ►3 Sum

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)

The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:

(-1, 0, 1)

(-1, -1, 2)

Solution: The key idea is the same as the TwoSum problem. When we fix the 1st number, the 2nd and3rd number can be found following the same reasoning as TwoSum.

The only difference is that, the TwoSum problem of LEETCODE has a unique solution. However, in ThreeSum, we have multiple duplicate solutions that can be found. Most of the OLE errors happened here because you could've ended up with a solution with so many duplicates.

The naive solution for the duplicates will be using the STL methods like below :

sort(res.begin(), res.end());

res.erase(unique(res.begin(), res.end()), res.end());

But according to my submissions, this way will cause you double your time consuming almostly.

A better approach is that, to jump over the number which has been scanned, no matter it is part of some solution or not.

If the three numbers formed a solution, we can safely ignore all the duplicates of them.

We can do this to all the three numbers such that we can remove the duplicates.

Here's my AC C++ Code:

class Solution {

public:

vector<vector<int> > threeSum(vector<int> &num) {

vector<vector<int> > res;

sort(num.begin(), num.end());

for (int i = 0; i < num.size(); i++) {

int target = -num[i];

int front = i + 1;

int back = num.size() - 1;

while (front < back) {

int sum = num[front] + num[back];

// Finding answer which start from number num[i]

if (sum < target)

front++;

else if (sum > target)

back--;

else {

vector<int> triplet(3, 0);

triplet[0] = num[i];

triplet[1] = num[front];

triplet[2] = num[back];

res.push\_back(triplet);

// Processing duplicates of Number 2

// Rolling the front pointer to the next different number forwards

while (front < back && num[front] == triplet[1]) front++;

// Processing duplicates of Number 3

// Rolling the back pointer to the next different number backwards

while (front < back && num[back] == triplet[2]) back--;

}

}

// Processing duplicates of Number 1

while (i + 1 < num.size() && num[i + 1] == num[i])

i++;

}

return res;

}

};

## ►Permutations

Given a collection of numbers, return all possible permutations.

For example,  
[1,2,3] have the following permutations:  
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

Failed cases:

Input:**[1]**

Expected:**[[1]]**

**class** Solution {

**public**:

vector<vector<**int**>> permute(vector<**int**>& nums) {

vector<vector<**int**>> res;

**int** n = nums.size();

**if**(n==0) **return** res;

vector<**bool**> flags(n, **true**);

vector<**int**> path;

permutations(nums, flags, res, path);

**return** res;

}

**void** permutations(vector<**int**>& nums, vector<**bool**> &flags,

vector<vector<**int**>> &res, vector<**int**> &path)

{

**int** i;

**bool** flag = **true**;

**for**(**bool** b:flags){

**if**(b) flag = **false**;

}

**if**(flag){

res.push\_back(path);

**return**;

}

**for**(i=0;i<nums.size();i++){

**if**(flags[i]==**true**){

flags[i] = **false**;

path.push\_back(nums[i]);

permutations(nums, flags, res, path);

flags[i] = **true**;

path.pop\_back();

}

}

}

};

**Solution 2:** Every Number chosen to be the gegin once

vector<vector<**int**> > permute(vector<**int**> &num) {

vector<vector<**int**> > ans;

permutation(num, 0, ans);

**return** ans;

}

**void** permutation(vector<**int**> &num, **int** begin, vector<vector<**int**> > &ans) {

**if** (begin >= num.size()) {

ans.push\_back(num);

**return**;

}

// every number chosen to be the begin once

**for** (**int** i = begin; i < num.size(); ++i) {

swap(num[begin], num[i]);

permutation(num, begin+1, ans);

swap(num[begin], num[i]);

}

}

## ►►Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,  
[1,1,2] have the following unique permutations:  
[1,1,2], [1,2,1], and [2,1,1].

**class** Solution {

**public**:

vector<vector<**int**>> permuteUnique(vector<**int**>& nums) {

vector<vector<**int**> > ans;

permutation(nums, 0, ans);

**return** ans;

}

**void** permutation(vector<**int**> &nums, **int** begin,

vector<vector<**int**> > &ans)

{

**if** (begin >= nums.size()) {

ans.push\_back(nums);

**return**;

}

unordered\_set<**int**> set;

// every number chosen to be the begin once

**for** (**int** i = begin; i < nums.size(); ++i) {

**if** (set.count(nums[i]) > 0) **continue**;

set.insert(nums[i]);

swap(nums[begin], nums[i]);

permutation(nums, begin+1, ans);

swap(nums[begin], nums[i]);

}

}

};

## ►►Permutation Sequence

The set [1,2,3,…,*n*] contains a total of *n*! unique permutations.

By listing and labeling all of the permutations in order,  
We get the following sequence (ie, for *n* = 3):

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given *n* and *k*, return the *k*th permutation sequence.

**Note:** Given *n* will be between 1 and 9 inclusive.

**Solution: Sometimes write down the input & output make you find the pattern quickly**

/\* We have to return the k-th permutation in lexicographic order. We can take a recursive approach. The lexicographic order means that first we have those starting with digit 1 then those starting with 2 and so on until digit n.

For example, for n = 4, we have:

k perm

--------

1 1234

2 1243

3 1324

4 1342

5 1423

6 1432

7 2134

8 2143

9 2314

10 2341

11 2413

12 2431

13 3124

14 3142

15 3214

16 3241

17 3412

18 3421

19 4123

20 4132

21 4213

22 4231

23 4312

24 4321

Since there are n digits, the number of all permutations starting with a given digit d is equal to (n-1)!. Given k, the permutation order number, we can determine which one is the first digit d: d = [(k-1)/(n-1)!]+1;

Once we determined the first digit we recursively determine the remaining digits. We keep all digits in a vector: {1,2,3,...,n}. Once we determine a digit we remove it from there and recursively solve the problem for the remaining n-1 digits. The formula above to determine the digit is written slightly different in code such that instead of the digit we first get its index in this vector.

From k we can also determine k for the n-1 problem as: prevK = k%prevFact. This is the order number within the current digit's group.

Now we can recursively solve the problem for n-1 and prevK and determine the rest of the digits.

\*/

**class** Solution

{

**int** factorial(**int** n){

**int** fact = 1;

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

fact \*= i;

**return** fact;

}

string getPermutation(**int** n, **int** k, vector<**int**>& digits)

{

**if** (n == 1) {

**return** to\_string(digits[0]);

}

**int** prevFact = factorial(n - 1);

**int** idx = (k - 1) / prevFact;

**int** d = digits[idx];

digits.erase(digits.begin() + idx);

**int** prevK = k%prevFact;

**if** (prevK == 0) prevK = prevFact;

string prev = getPermutation(n - 1, prevK, digits);

**return** to\_string(d) + prev;

}

**public**:

string getPermutation(**int** n, **int** k)

{

vector<**int**> digits;

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

digits.push\_back(i);

}

**return** getPermutation(n, k, digits);

}

};

## Multiply Strings

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.

Failed cases:

Input:**"0", "0"**

Expected:**"1"**

Input:**"1", "1"**

Expected:**"1"**

**class** Solution {

**public**:

string multiply(string num1, string num2) {

string res;

**int** i;

reverse(num1.begin(), num1.end());

reverse(num2.begin(), num2.end());

**for**(i=0;i<num2.length();i++){

string product = multiply(num1, num2[i], i);

res = add(res, product);

}

**for**(i=res.length()-1;i>=1;i--){

**if**(res[i]=='0') res.resize(res.length() - 1);

**else** **break**;

}

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

**return** res;

}

string multiply(string num1, **char** num2, **int** off){

string res;

**int** i;

**int** carry = 0;

**int** product;

**if**(num2 == '0') **return** string("0");

**for**(i=0;i<off;i++) res += '0';

**for**(i=0;i<num1.size();i++){

product = (num1[i]-'0')\*(num2-'0') + carry;

carry = product/10;

product %= 10;

res += '0' + product;

}

**if**(carry!=0) res += '0' + carry;

**return** res;

}

string add(string num1, string num2){

string res;

**int** i,carry=0;

**int** sum = 0;

**int** n1,n2;

**for**(i=0;i<max(num1.length(),num2.length());i++){

**if**(i<num1.length()) n1 = num1[i]-'0' ;

**else** n1 = 0;

**if**(i<num2.length()) n2 = num2[i]-'0' ;

**else** n2 = 0;

sum = n1 + n2 + carry;

carry = sum/10;

sum %= 10;

res += '0' + sum;

}

**if**(carry!=0) res += '0' + carry;

**return** res;

}

};

Solution 2: Brief C++ solution using only strings and without reversal

This is the standard manual multiplication algorithm. We use two nested for loops, working backward from the end of each input number. We pre-allocate our result and accumulate our partial result in there. One special case to note is when our carry requires us to write to our sum string outside of our for loop.

At the end, we trim any leading zeros, or return 0 if we computed nothing but zeros.

**class** Solution {

**public**:

string multiply(string num1, string num2) {

string sum(num1.size() + num2.size(), '0');

**for** (**int** i = num1.size() - 1; 0 <= i; --i) {

**int** carry = 0;

**for** (**int** j = num2.size() - 1; 0 <= j; --j) {

**int** tmp = (sum[i + j + 1] - '0') + (num1[i] - '0') \* (num2[j] - '0') + carry;

sum[i + j + 1] = tmp % 10 + '0';

carry = tmp / 10;

}

sum[i] += carry;

}

size\_t startpos = sum.find\_first\_not\_of("0");

**if** (string::npos != startpos) {

**return** sum.substr(startpos);

}

**return** "0";

}

};

## ►Pow(x, n)

 Implement pow(*x*, *n*).

Fialed Cases:

Last executed input:**34.00515, -3**

**class** Solution {

**public**:

**double** myPow(**double** x, **int** n) {

**if** (n == 0) **return** 1;

**if** (n < 0) {

x = 1 / x;

n = -n;

}

**double** res = myPow(x, n / 2);

**if**(n%2==0) **return** res\*res;

**if**(n%2!=0) **return** res\*res\*x;

}

};

## ►Majority Element

Given an array of size *n*, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

Solution: Boyer-Moore Majority Vote algorithm

**class** Solution {

**public**:

**int** majorityElement(vector<**int**>& nums) {

**int** candidate = 0;

**int** count = 0;

**for**(**int** i : nums){

**if**(candidate == i) count ++;

**else** **if**(count == 0) {

candidate = i;

count = 1;

}**else** count --;

}

**return** candidate;

}

};

## ►►Majority Element II

Given an integer array of size *n*, find all elements that appear more than ⌊ n/3 ⌋ times. The algorithm should run in linear time and in O(1) space.

Solution: For those who aren't familiar with Boyer-Moore Majority Vote algorithm, I found a great article (http://goo.gl/64Nams) that helps me to understand this fantastic algorithm!! Please check it out!

The essential concepts is you keep a counter for the majority number **X**. If you find a number **Y**that is not **X**, the current counter should deduce 1. The reason is that if there is 5 **X** and 4 **Y**, there would be one (5-4) more **X** than **Y**. This could be explained as "4 **X** being paired out by 4**Y**".

And since the requirement is finding the majority for more than ceiling of [n/3], the answer would be less than or equal to two numbers. So we can modify the algorithm to maintain two counters for two majorities.

Followings are my sample Python code:

**class** Solution {

**public**:

vector<**int**> majorityElement(vector<**int**>& nums) {

**int** count1 = 0, count2 = 0;

**int** candidate1 = 0, candidate2 = 1;

**for**(**int** n: nums){

**if**(candidate1 == n) count1 ++;

**else** **if**(candidate2 == n) count2 ++;

**else** **if** (count1 == 0){

count1 = 1;

candidate1 = n;

}**else** **if** (count2 == 0 ){

count2 = 1;

candidate2 = n;

} **else**{

count1--; count2--;

}

}

count1 = count2 = 0;

**for**(**int** n: nums){

**if**(n == candidate1) count1++;

**else** **if**(n == candidate2) count2++;

}

vector<**int**> result;

**if** (count1 > nums.size()/3) result.push\_back(candidate1);

**if** (count2 > nums.size()/3) result.push\_back(candidate2);

**return** result;

}

};

## ►►►Valid Number (DFS , Regular Expression)

Validate if a given string is numeric.

Some examples:  
"0" => true  
" 0.1 " => true  
"abc" => false  
"1 a" => false  
"2e10" => true

**Note:** It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

**Solution 1: Regular Expression**

There are two possible expressions for this problem:

1. bs\* sign \d+ [dot \d] [e sign \d+] bs
2. bs\* sign [\d] dot \d+ [e sign \d+] bs

The part in [ ... ] is alternative.

**class** Solution {

**public**:

**void** getBs(string& s, **int**& i){

**while**(i<s.length() && s[i] == ' ')

++i;

}

**void** getSign(string& s, **int** &i){

**if**(i<s.length() && (s[i]=='+' || s[i]=='-'))

i++;

}

**bool** getDigit(string& s, **int**&i){

**int** j=i;

**while**(i<s.length() && isdigit(s[i]))

++i;

**return** (i!=j);

}

**bool** getDot(string& s, **int**&i){

**if**(i<s.length() && s[i]=='.')

i++;

}

**bool** getE(string& s, **int**&i){

**if**(i<s.length() && (s[i]=='e' || s[i]=='E'))

i++;

}

**bool** isNumber(string s) {

**int** i=0;

**bool** integer=**false**,decimal=**false**;

getBs(s, i);

getSign(s,i);

**if**(getDigit(s,i)){

integer = **true**;

}

**if**(getDot(s,i)){

**if**(getDigit(s,i))

decimal = **true**;

}

**if**(getE(s,i)){

getSign(s,i);

**if**(!getDigit(s,i)) **return** **false**;

}

getBs(s,i);

**if**(i!= s.length()) **return** **false**;

**if**(integer == **false** && decimal==**false**)

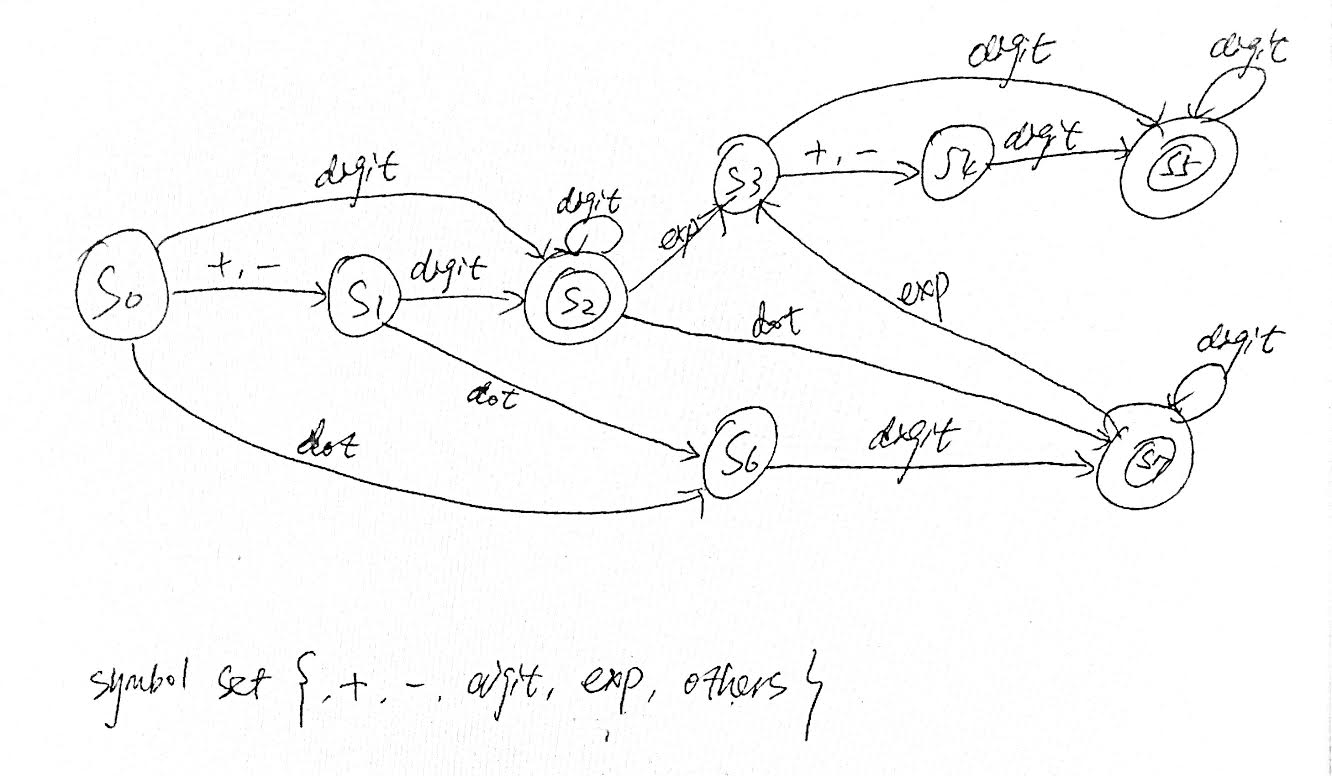
**return** **false**;

**return** **true**;

}

};

**Solution 2: DFA**

****

There are **5** kind of inputs in my DFA:

digit : number **0-9** for

+,- : operator **+** or **-**(negative or positive)

exp: **e**

dot: **.**

other: you can return **false** Immediately

**4 final States** in my DFA transition diagram : **s2, s6, s7, s8**

**If the state change to final state at last, return true.**

s2 can accept digits only : +1 -23432 123 and etc

s5 can accept exp expression: +2.4e+12 3e9 and etc

s6 can accept decimals end with dot: 1. -42. and etc (careful, what if there exist only one dot "." \*\*\*\* I use a variable flag judging weather there existing numbers. cause 0. and .0 is valid and . is invalid \*\*\*\* )

s7 can accept decimals: +12.23, 87., 132

It is clear how DFA works in my pictures. We just need to handle the inputs, and update the state according to DFA

**class** Solution {

**public**:

**bool** isNumber(string str) {

**int** state=0

**int** flag=0; //flag to judge the special case "."

**while**(str[0]==' ') str.erase(0,1);

**while**(str[str.length()-1]==' ')

str.erase(str.length()-1, 1);

**for**(**int** i=0; i<str.length(); i++){

**if**('0'<=str[i] && str[i]<='9'){

flag=1;

**if**(state<=2) state=2;

**else** state=(state<=5)?5:7;

}

**else** **if**('+'==str[i] || '-'==str[i]){

**if**(state==0 || state==3) state++;

**else** **return** **false**;

}

**else** **if**('.'==str[i]){

**if**(state<=2) state=6;

**else** **return** **false**;

}

**else** **if**('e'==str[i]){

**if**(flag&&(state==2||state==6||state==7))

state=3;

**else** **return** **false**;

}

**else** **return** **false**;

}

**return** (state==2 || state==5 ||

(flag&&state==6) || state==7);

}

};

# Best Time to Buy and Sell Stock

## ►Best Time to Buy and Sell Stock I

Say you have an array for which the ith element is the price of a given stock on day i. If you were only permitted to complete at most **one** transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

**class** Solution {

**public**:

**int** maxProfit(vector<**int**>& prices) {

**int** minPrice = INT\_MAX;

**int** maxProfit = 0;

**int** i,p;

**for**(i=0;i<prices.size();i++){

p = prices[i];

**if**(p<minPrice) minPrice = p;

**if**(p-minPrice>maxProfit) maxProfit = p - minPrice;

}

**return** maxProfit;

}

};

## ►►Best Time to Buy and Sell Stock II

Say you have an array for which the *i*th element is the price of a given stock on day *i*. Design an algorithm to find the maximum profit. You may complete **as many transactions as you like** (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution: **Greedy**

**class** Solution {

**public**:

**int** maxProfit(vector<**int**>& prices) {

**int** res = 0;

**int** n = prices.size();

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

**if**(prices[i+1]>prices[i]){

res += prices[i+1] - prices[i];

}

}

**return** res;

}

};

## ►►► Best Time to Buy and Sell Stock III

Say you have an array for which the *i*th element is the price of a given stock on day *i*. Design an algorithm to find the maximum profit. You may complete at most ***two*** transactions.

**Note:**You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution: **Dynamic Programming**

**class** Solution {

**public**:

**int** maxProfit(vector<**int**>& prices) {

**int** n = prices.size();

**if**(n==0) **return** 0;

**int** maxPro = 0;

**int** i;

vector<**int**> preProfit(n,0);

vector<**int**> postProfit(n,0);

preProfit[0] = 0;

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

preProfit[i] = prices[i]>prices[i-1] ?

maxProfit(prices, 0, i) : preProfit[i-1];

maxPro = max(maxPro, preProfit[i]);

}

postProfit[n-1] = 0;

**for**(i=n-2;i>=0;i--){

postProfit[i] = prices[i]<prices[i+1] ?

maxProfit(prices, i, n-1) : postProfit[i+1];

maxPro = max(maxPro, postProfit[i]);

}

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

maxPro =max(maxPro, preProfit[i]+postProfit[i+1]);

**return** maxPro;

}

**int** maxProfit(vector<**int**>& prices,**int** s, **int** e) {

**int** minPrice = INT\_MAX;

**int** maxProfit = 0;

**int** i,p;

**for**(i=s;i<=e;i++){

p = prices[i];

**if**(p<minPrice) minPrice = p;

**if**(p-minPrice>maxProfit) maxProfit = p - minPrice;

}

**return** maxProfit;

}

};

# Two Pointers, Sliding Window

## Container With Most Water

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

28ms Solution:

**class** Solution {

**public**:

**int** maxArea(vector<**int**>& height) {

**int** n = height.size();

**if**(n==0) **return** 0;

**int** res = INT\_MIN;

**int** i=0, j=n-1;

**while**(i<j){

**while**(height[i]<height[j] && i<j){

res = max(res, (j-i)\*height[i]);

i++;

}

**while**(height[i]>=height[j] && i<j){

res = max(res, (j-i)\*height[j]);

j--;

}

}

**return** res;

}

};

40ms Solution: Start by evaluating the widest container, using the first and the last line. All other possible containers are less wide, so to hold more water, they need to be higher. Thus, after evaluating that widest container, skip lines at both ends that don't support a higher height. Then evaluate that new container we arrived at. Repeat until there are no more possible containers left.

**int** maxArea(vector<**int**>& height) {

**int** water = 0;

**int** i = 0, j = height.size() - 1;

**while** (i < j) {

**int** h = min(height[i], height[j]);

water = max(water, (j - i) \* h);

**while** (height[i] <= h && i < j) i++;

**while** (height[j] <= h && i < j) j--;

}

**return** water;

}

►Minimum Size Subarray Sum

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum ≥ s. If there isn't one, return 0 instead.

For example, given the array [2,3,1,2,4,3] and s = 7,  
the subarray [4,3] has the minimal length under the problem constraint.

Hint: Subarray imply continues subarray.

class Solution {

public:

int minSubArrayLen(int s, vector<int>& nums) {

int firstPos = 0, sum = 0, minLength = INT\_MAX;

for(int i = 0; i<nums.size(); i++) {

sum += nums[i];

while(sum >= s) {

minLength = min(minLength, i - firstPos + 1);

sum -= nums[firstPos++];

}

}

return minLength == INT\_MAX? 0 : minLength;

}

};

## ►Sort Colors

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

**Note:**  
You are not suppose to use the library's sort function for this problem.

**Follow up:**  
A rather straight forward solution is a two-pass algorithm using counting sort.  
First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

**Solution:** Maintain the tail index for red region, and head index for the blue region. Scan the whole array, and swap the current element with either red tail or blue head respectively.

**class** Solution {

**public**:

**void** sortColors(vector<**int**>& nums) {

**int** tail\_red = 0;

**int** head\_blue = nums.size() - 1;

**int** cur = 0;

**while**( cur <= head\_blue){

**if**(nums[cur] == 0){

swap(nums[tail\_red], nums[cur]);

tail\_red ++;

cur ++;

}**else** **if** (nums[cur] == 2){

swap(nums[head\_blue], nums[cur]);

head\_blue = head\_blue - 1;

}**else**

cur ++;

}

}

};

## ►►►Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,   
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.



Solution: Tow Pointors

Keep track of the already safe level and the total water so far. In each step, process and discard the lower one of the leftmost or rightmost elevation.

**class** Solution {

**public**:

**int** trap(vector<**int**>& height) {

**int** l = 0, r = height.size()-1;

**int** level = 0, water = 0;

**while** (l < r) {

**int** lower = height[height[l] < height[r] ? l++ : r--];

level = max(level, lower);

water += level - lower;

}

**return** water;

}

};

## ►►►Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,  
**S** = "ADOBECODEBANC"  
**T** = "ABC"

Minimum window is "BANC".

**Note:**  
If there is no such window in S that covers all characters in T, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

**class** Solution {

**public**:

string minWindow(string s, string t) {

vector<**int**> remaining(128,0);

**int** i=0, j=-1, k=0;

**int** n = s.length();

**int** m = t.length();

string res = "";

**int** minStart, minL = INT\_MAX;

**if**(n==0) **return** res;

**for**(**char** c:t){

remaining[c] ++;

}

//init the sliding window s[i,j]

//make it complete

**bool** isComplete = **false**;

**while**(!isComplete && j<n){

j++;

remaining[s[j]]--;

**while**(remaining[t[k]]<=0 && k<m) k++;

**if**(k==m) isComplete = **true**;

}

**while**(j<n){

**while**(!isComplete && j<n){

j++;

remaining[s[j]]--;

**if**(i>=1 && s[i-1]==s[j]) isComplete = **true**;

}

**while**(isComplete && i<=j){

**if**(minL>j-i+1){

minL = j-i+1;

minStart = i;

}

remaining[s[i]] ++;

**if**(remaining[s[i]] > 0) isComplete = **false**;

i++;

}

}

**if**(minL!=INT\_MAX) res = s.substr(minStart, minL);

**return** res;

}

};

# Stack

## Valid Parentheses

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

**class** Solution {

**public**:

**bool** isValid(string s) {

stack<**char**> st;

**char** t;

**if**(s.length()%2 != 0) **return** **false**;

**for**(**char** c:s){

**if**(c=='('||c=='{'||c=='[') st.push(c);

**else** **if**(!st.empty()) {

t = st.top();

**if**((t=='(' && c==')') ||

(t=='{' && c=='}') ||

(t=='[' && c==']' ))

{

st.pop();

}**else** **return** **false**;

}**else** **return** **false**;

}

**return** st.empty();

}

};

## ►►►Longest Valid Parentheses

Given a string containing just the characters '(' and')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

**Failed cases:**

Input:**"()(()"**

Output:**4**

Expected:**2**

Input:**"()()"**

Output:**2**

Expected:**4**

Input:**")()()("**

Output:**2**

Expected:**4**

**class** Solution {

**public**:

**int** longestValidParentheses(string s) {

**int** ret = 0;

stack<**int**> st;

st.push(-1);

**for**(**int** i=0;i<s.length();i++){

**int** t = st.top();

**if**(t!=-1 && s[t]=='(' && s[i]==')') {

st.pop();

ret = max(ret, i-st.top());

}**else** st.push(i);

}

**return** ret;

}

};

## ►►►Largest Rectangle in Histogram

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

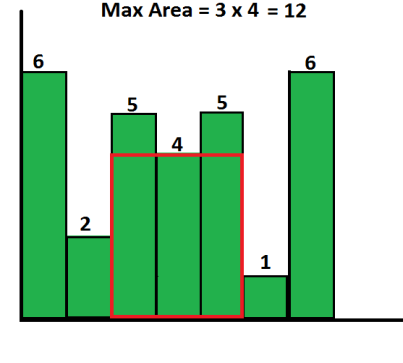


The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example,  
Given height = [2,1,5,6,2,3],  
return 10.

**Solution on Geeks4geeks:**

<http://www.geeksforgeeks.org/largest-rectangle-under-histogram/>

[](http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/histogram1.png)

For every bar ‘x’, we calculate the area with ‘x’ as the smallest bar in the rectangle. If we calculate such area for every bar ‘x’ and find the maximum of all areas, our task is done. How to calculate area with ‘x’ as smallest bar? We need to know index of the first smaller (smaller than ‘x’) bar on left of ‘x’ and index of first smaller bar on right of ‘x’. Let us call these indexes as ‘left index’ and ‘right index’ respectively.  
We traverse all bars from left to right, maintain a stack of bars. Every bar is pushed to stack once. A bar is popped from stack when a bar of smaller height is seen. When a bar is popped, we calculate the area with the popped bar as smallest bar. How do we get left and right indexes of the popped bar – the current index tells us the ‘right index’ and index of previous item in stack is the ‘left index’.

**Solution 1:**

**1)**Create an empty stack.

**2)**Start from first bar, and do following for every bar ‘height[i]’ where ‘i’ varies from 0 to n-1.

**a)** If stack is empty or height[i] is higher than the bar at top of stack, then push ‘i’ to stack.  
 **b)** If this bar is smaller than the top of stack, then pop up the top of stack. Let the removed bar be height[tp]. Calculate area of rectangle with height[tp] as smallest bar. For height[tp], the ‘left index’ is previous (current top of stack) item in stack and ‘right index’ is ‘i’ (current index).

**int** largestRectangleArea(vector<**int**> &height) {

// push a sentinel node back into the end of height

// to make the code logic more concise

height.push\_back(0);

stack<**int**> s;

**int** ret = 0, h = 0, w = 0;

**int** i = 0, n = height.size();

**while** (i < n) {

//If this bar is higher than the bar on top stack

//push it to stack

**if** (s.empty() || height[s.top()] <= height[i])

s.push(i++);

//If this bar is lower than top of stack,

//then calculate area of rectangle with stack top

//as the smallest (or minimum height) bar.

//'i' is 'right index' for the top

//and element before top in stack is 'left index'

**else** {

h = height[s.top()];

s.pop();

w = s.empty() ? i : i - s.top() - 1;

**if** (h \* w > ret) ret = h \* w;

}

}

**return** ret;

}

**Solution 2:**

// all the nodes just push in and pop out once

// time complexity is O(2n)

**int** largestRectangleArea(vector<**int**> &height) {

**int** ret = 0;

height.push\_back(0);

stack<**int**> s;

**int** h = 0, w = 0;

**for**(**int** i = 0; i < height.size(); i++) {

**while**(!s.empty() && height[s.top()] >= height[i]) {

h = height[s.top()];

s.pop();

w = s.empty() ? i : i - s.top() - 1;

**if**(h \* w > ret) ret = h \* w;

}

s.push(i);

}

**return** ret;

}

## ►►►►Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

**Solution:** Same Idea with “Largest Rectangle in Histogram”

**int** maximalRectangle(vector<vector<**char**> > &matrix) {

**if** (matrix.size() <= 0 || matrix[0].size() <= 0)

**return** 0;

**int** m = matrix.size();

**int** n = matrix[0].size() + 1;

**int** h = 0, w = 0, ret = 0;

vector<**int**> height(n, 0);

**for** (**int** i = 0; i < m; ++i) {

stack<**int**> s;

**for** (**int** j = 0; j < n; ++j) {

// set value

**if** (j < n - 1) {

**if** (matrix[i][j] == '1') height[j] += 1;

**else** height[j] = 0;

}

// compute area

**while** (!s.empty() && height[s.top()] >= height[j]) {

h = height[s.top()];

s.pop();

w = s.empty() ? j : j - s.top() - 1;

**if** (h \* w > ret) ret = h \* w;

}

s.push(j);

}

}

**return** ret;

}

# [Thinking Recursively](http://www.youtube.com/watch?v=uFJhEPrbycQ&feature=PlayList&p=FE6E58F856038C69&index=8)

## ►Search in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

*Solution: For simple if rotated then search both pre subarray and post subarray.*

**class** Solution {

**public**:

**int** search(vector<**int**>& nums, **int** target) {

**return** search(nums, 0, nums.size()-1, target);

}

**int** search(vector<**int**>& nums,

**int** i, **int** j, **int** target) {

**int** n = j-i+1;

**if**(n<1) **return** -1;

**if**(n==1) **return** nums[i]==target?i:-1;

**int** pre\_ret =-1, post\_ret = -1;

**int** mid=(i+j)/2;

**if**(nums[mid]==target) **return** mid;

**else** **if**(nums[mid]>target){

pre\_ret = search(nums,i,mid-1,target);

**if**(pre\_ret!=-1) **return** pre\_ret;

**if**(nums[i]>nums[j])

post\_ret = search(nums,mid+1,j,target);

**return** max(pre\_ret, post\_ret);

}**else**{

post\_ret = search(nums,mid+1,j,target);

**if**(post\_ret!=-1) **return** post\_ret;

**if**(nums[i]>nums[j])

pre\_ret = search(nums,i,mid-1,target);

**return** max(pre\_ret, post\_ret);

}

}

};

## Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":  
What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Test case:

Input: [1, 3, 1, 1, 1, 1, 1] 3

Output: true

Solution: Only change the condition of rotated of not from

**if**(nums[i]>nums[j]) to **if**(nums[i]>=nums[j]);

**class** Solution {

**public**:

**bool** search(vector<**int**>& nums, **int** target) {

**return** search(nums, 0, nums.size()-1, target);

}

**bool** search(vector<**int**>& nums,

**int** i, **int** j, **int** target) {

**int** n = j-i+1;

**if**(n<1) **return** **false**;

**if**(n==1) **return** nums[i]==target;

**bool** pre\_ret =**false**, post\_ret = **false**;

**int** mid=(i+j)/2;

**if**(nums[mid]==target) **return** **true**;

**else** **if**(nums[mid]>target){

pre\_ret = search(nums,i,mid-1,target);

**if**(pre\_ret) **return** **true**;

**if**(nums[i]>=nums[j])

post\_ret = search(nums,mid+1,j,target);

**return** post\_ret;

}**else**{

post\_ret = search(nums,mid+1,j,target);

**if**(post\_ret) **return** **true**;

**if**(nums[i]>=nums[j])

pre\_ret = search(nums,i,mid-1,target);

**return** pre\_ret;

}

}

};

## Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

If the target is not found in the array, return [-1, -1].

For example,  
Given [5, 7, 7, 8, 8, 10] and target value 8,  
return [3, 4].

**class** Solution {

**public**:

vector<**int**> searchRange(vector<**int**>& nums, **int** target) {

**return** searchRange(nums,0,nums.size()-1,target);

}

vector<**int**> searchRange(vector<**int**>& nums,

**int** i, **int** j, **int** target) {

vector<**int**> ret(2,-1);

**int** n = j-i+1;

**if**(n<1) **return** ret;

**if**(n==1){

**if**(nums[i]==target){

ret[0] = i;

ret[1] = j;

}

**return** ret;

}

**int** mid = (j+i)/2;

**if**(nums[mid]==target){

vector<**int**> pre\_ret,post\_ret;

pre\_ret = searchRange(nums,i,mid,target);

post\_ret= searchRange(nums,mid+1,j,target);

**if**(pre\_ret[0]==-1) **return** post\_ret;

**else** **if**(post\_ret[0]==-1) **return** pre\_ret;

**else**{

pre\_ret[1] = post\_ret[1];

**return** pre\_ret;

}

}**else** **if**(nums[mid]>target){

**return** searchRange(nums,i,mid,target);

}**else**{

**return** searchRange(nums,mid+1,j,target);

}

}

};

## ►Unique Binary Search Trees II

 Given *n*, generate all structurally unique **BST's** (binary search trees) that store values 1...*n*.

For example,  
Given *n* = 3, your program should return all 5 unique BST's shown below.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

vector<TreeNode\*> generateTrees(**int** n) {

**return** generateSubtrees(1, n);

}

vector<TreeNode\*> helper(**int** s, **int** e) {

vector<TreeNode\*> result;

**if** (s > e) {

result.push\_back(NULL); // empty tree

**return** result;

}

**for** (**int** i = s; i <= e; ++i) {

vector<TreeNode\*> leftSubtrees = helper(s, i - 1);

vector<TreeNode\*> rightSubtrees = helper(i + 1, e);

**for** (TreeNode\* left : leftSubtrees) {

**for** (TreeNode\* right : rightSubtrees) {

TreeNode\* root = **new** TreeNode(i);

root->left = left;

root->right = right;

result.push\_back(root);

}

}

}

**return** result;

}

## Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following is not:

1

/ \

2 2

\ \

3 3

class Solution {

public:

bool isSymmetric(TreeNode \*root) {

if(root == NULL) return true;

return isSymmetric(root->left,root->right);

}

bool isSymmetric(TreeNode \*p, TreeNode\* q) {

if(p==NULL && q==NULL) return true;

if(p==NULL && q!=NULL || p!=NULL && q==NULL) return false;

if(p->val == q->val && isSymmetric(p->left, q->right) && isSymmetric(p->right, q->left)) {

return true;

}

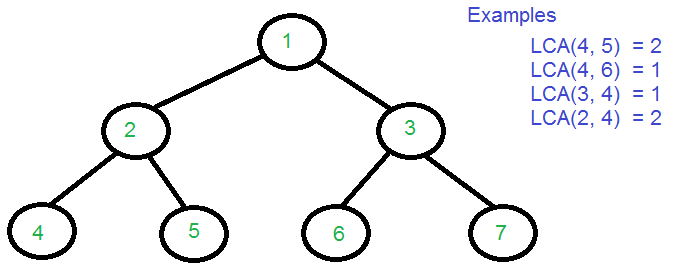
return false;

}

};

## ►LCA -- Lowest Common Ancestor in a Binary Tree

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/lca.png)

Method 1 (By Storing root to n1 and root to n2 paths):  
Following is simple O(n) algorithm to find LCA of n1 and n2.  
1) Find path from root to n1 and store it in a vector or array.  
2) Find path from root to n2 and store it in another vector or array.  
3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

Following is C++ implementation of above algorithm.

|  |
| --- |
| // A O(n) solution to find LCA of two given values n1 and n2  #include <iostream>  #include <vector>  using namespace std;    // A Bianry Tree node  struct Node  {      int key;      struct Node \*left, \*right;  };    // Utility function creates a new binary tree node with given key  Node \* newNode(int k)  {      Node \*temp = new Node;      temp->key = k;      temp->left = temp->right = NULL;      return temp;  }    // Finds the path from root node to given root of the tree, Stores the  // path in a vector path[], returns true if path exists otherwise false  bool findPath(Node \*root, vector<int> &path, int k)  {      // base case      if (root == NULL) return false;        // Store this node in path vector. The node will be removed if      // not in path from root to k      path.push\_back(root->key);        // See if the k is same as root's key      if (root->key == k)          return true;        // Check if k is found in left or right sub-tree      if ( (root->left && findPath(root->left, path, k)) ||           (root->right && findPath(root->right, path, k)) )          return true;        // If not present in subtree rooted with root, remove root from      // path[] and return false      path.pop\_back();      return false;  }    // Returns LCA if node n1, n2 are present in the given binary tree,  // otherwise return -1  int findLCA(Node \*root, int n1, int n2)  {      // to store paths to n1 and n2 from the root      vector<int> path1, path2;        // Find paths from root to n1 and root to n1. If either n1 or n2      // is not present, return -1      if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))            return -1;        /\* Compare the paths to get the first different value \*/      int i;      for (i = 0; i < path1.size() && i < path2.size() ; i++)          if (path1[i] != path2[i])              break;      return path1[i-1];  }    // Driver program to test above functions  int main()  {      // Let us create the Binary Tree shown in above diagram.      Node \* root = newNode(1);      root->left = newNode(2);      root->right = newNode(3);      root->left->left = newNode(4);      root->left->right = newNode(5);      root->right->left = newNode(6);      root->right->right = newNode(7);      cout << "LCA(4, 5) = " << findLCA(root, 4, 5);      cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);      cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);      cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);      return 0;  } |

Output:

LCA(4, 5) = 2

LCA(4, 6) = 1

LCA(3, 4) = 1

LCA(2, 4) = 2

Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.  
Thanks to Ravi Chandra Enaganti for suggesting the initial solution based on this method.

**Method 2 (Using Single Traversal)**  
The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.  
The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn’t match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

|  |
| --- |
| /\* Program to find LCA of n1 and n2 using one traversal of Binary Tree \*/  #include <iostream>  using namespace std;    // A Binary Tree Node  struct Node  {      struct Node \*left, \*right;      int key;  };    // Utility function to create a new tree Node  Node\* newNode(int key)  {      Node \*temp = new Node;      temp->key = key;      temp->left = temp->right = NULL;      return temp;  }    // This function returns pointer to LCA of two given values n1 and n2.  // This function assumes that n1 and n2 are present in Binary Tree  struct Node \*findLCA(struct Node\* root, int n1, int n2)  {      // Base case      if (root == NULL) return NULL;        // If either n1 or n2 matches with root's key, report      // the presence by returning root (Note that if a key is      // ancestor of other, then the ancestor key becomes LCA  if (root->key == n1 || root->key == n2)        return root;        // Look for keys in left and right subtrees      Node \*left\_lca  = findLCA(root->left, n1, n2);      Node \*right\_lca = findLCA(root->right, n1, n2);        // If both of the above calls return Non-NULL, then one key      // is present in once subtree and other is present in other,      // So this node is the LCA      if (left\_lca && right\_lca)  return root;        // Otherwise check if left subtree or right subtree is LCA      return (left\_lca != NULL)? left\_lca: right\_lca;  }    // Driver program to test above functions  int main()  {      // Let us create binary tree given in the above example      Node \* root = newNode(1);      root->left = newNode(2);      root->right = newNode(3);      root->left->left = newNode(4);      root->left->right = newNode(5);      root->right->left = newNode(6);      root->right->right = newNode(7);      cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;      cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6)->key;      cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4)->key;      cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4)->key;      return 0;  } |

Output:

LCA(4, 5) = 2

LCA(4, 6) = 1

LCA(3, 4) = 1

LCA(2, 4) = 2

Thanks to Atul Singh for suggesting this solution.

Time Complexity: Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion.  
Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL).  
We can extend this method to handle all cases by passing two boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

|  |
| --- |
| /\* Program to find LCA of n1 and n2 using one traversal of Binary Tree.     It handles all cases even when n1 or n2 is not there in Binary Tree \*/  #include <iostream>  using namespace std;    // A Binary Tree Node  struct Node  {      struct Node \*left, \*right;      int key;  };    // Utility function to create a new tree Node  Node\* newNode(int key)  {      Node \*temp = new Node;      temp->key = key;      temp->left = temp->right = NULL;      return temp;  }    // This function returns pointer to LCA of two given values n1 and n2.  // v1 is set as true by this function if n1 is found  // v2 is set as true by this function if n2 is found  struct Node \*findLCAUtil(struct Node\* root, int n1, int n2, bool &v1, bool &v2)  {      // Base case      if (root == NULL) return NULL;        // If either n1 or n2 matches with root's key, report the presence      // by setting v1 or v2 as true and return root (Note that if a key      // is ancestor of other, then the ancestor key becomes LCA)      if (root->key == n1)      {          v1 = true;          return root;      }      if (root->key == n2)      {          v2 = true;          return root;      }        // Look for keys in left and right subtrees      Node \*left\_lca  = findLCAUtil(root->left, n1, n2, v1, v2);      Node \*right\_lca = findLCAUtil(root->right, n1, n2, v1, v2);        // If both of the above calls return Non-NULL, then one key      // is present in once subtree and other is present in other,      // So this node is the LCA      if (left\_lca && right\_lca)  return root;        // Otherwise check if left subtree or right subtree is LCA      return (left\_lca != NULL)? left\_lca: right\_lca;  }    // Returns true if key k is present in tree rooted with root  bool find(Node \*root, int k)  {      // Base Case      if (root == NULL)          return false;        // If key is present at root, or in left subtree or right subtree,      // return true;      if (root->key == k || find(root->left, k) ||  find(root->right, k))          return true;        // Else return false      return false;  }    // This function returns LCA of n1 and n2 only if both n1 and n2 are present  // in tree, otherwise returns NULL;  Node \*findLCA(Node \*root, int n1, int n2)  {      // Initialize n1 and n2 as not visited      bool v1 = false, v2 = false;        // Find lca of n1 and n2 using the technique discussed above      Node \*lca = findLCAUtil(root, n1, n2, v1, v2);        // Return LCA only if both n1 and n2 are present in tree      if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))          return lca;        // Else return NULL      return NULL;  }    // Driver program to test above functions  int main()  {      // Let us create binary tree given in the above example      Node \* root = newNode(1);      root->left = newNode(2);      root->right = newNode(3);      root->left->left = newNode(4);      root->left->right = newNode(5);      root->right->left = newNode(6);      root->right->right = newNode(7);      Node \*lca =  findLCA(root, 4, 5);      if (lca != NULL)         cout << "LCA(4, 5) = " << lca->key;      else         cout << "Keys are not present ";        lca =  findLCA(root, 4, 10);      if (lca != NULL)         cout << "\nLCA(4, 10) = " << lca->key;      else         cout << "\nKeys are not present ";        return 0;  } |

Output:

LCA(4, 5) = 2

Keys are not present

Thanks to Dhruv for suggesting this extended solution.

We will soon be discussing more solutions to this problem. Solutions considering the following.  
1) If there are many LCA queries and we can take some extra preprocessing time to reduce the time taken to find LCA.  
2) If parent pointer is given with every node.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Spiral Matrix

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

For example,  
Given the following matrix:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

You should return [1,2,3,6,9,8,7,4,5].

**Corner Cases:**

[ [ 1, 2, 3 ] ];

[

[1],

[4],

[7]

]

[[1]];

**class** Solution {

**public**:

vector<**int**> spiralOrder(vector<vector<**int**>>& matrix) {

vector<**int**> res;

**int** m = matrix.size();

**if**(m==0) **return** res;

**int** n = matrix[0].size();

spiralOrder(matrix, 0, 0, m-1, n-1, res);

**return** res;

}

**void** spiralOrder(vector<vector<**int**>>& matrix, **int** x, **int** y,

**int** m, **int** n, vector<**int**> &res)

{

**int** i,j;

**if**(x==m && y==n){

res.push\_back(matrix[x][y]);

**return**;

}

**for**(i=x;i<n;i++) res.push\_back(matrix[x][i]);

**for**(i=x;i<m;i++) res.push\_back(matrix[i][n]);

**if**(x!=m){

**for**(i=n;i>y;i--) res.push\_back(matrix[m][i]);

}**else** res.push\_back(matrix[x][n]);

**if**(y!=n) {

**for**(i=m;i>x;i--) res.push\_back(matrix[i][y]);

}**else** res.push\_back(matrix[m][n]);

x++; m--; y++; n--;

**if**(m>=x && n>=y) spiralOrder(matrix,x,y,m,n,res);

}

};

## Spiral Matrix II

Given an integer *n*, generate a square matrix filled with elements from 1 to *n*2 in spiral order.

For example,  
Given *n* = 3,

You should return the following matrix:

[

[ 1, 2, 3 ],

[ 8, 9, 4 ],

[ 7, 6, 5 ]

]

**class** Solution {

**public**:

vector<vector<**int**>> generateMatrix(**int** n) {

vector<vector<**int**>> matrix(n,vector<**int**> (n, 0));

**if**(n==0) **return** matrix;

**int** count = 1;

generateMatrix(0, n-1, count, matrix);

**return** matrix;

}

**void** generateMatrix(**int** x, **int** n, **int** &count, vector<vector<**int**>> &matrix){

**int** i,j;

**if**(x == n){

matrix[x][x] = count++;

**return**;

}

**for**(i=x; i<n; i++) matrix[x][i] = count++;

**for**(i=x; i<n; i++) matrix[i][n] = count++;

**if**(x != n){

**for**(i=n;i>x;i--) matrix[n][i] = count++;

}**else** matrix[x][n] = count++;

**if**(x != n){

**for**(i=n;i>x;i--) matrix[i][x] = count++;

}**else** matrix[n][n] = count++;

x++; n--;

**if**(n>=x) generateMatrix(x, n, count, matrix);

}

};

## Rotate Image

You are given an *n* x *n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:  
Could you do this in-place?

**class** Solution {

**public**:

**void** rotate(vector<vector<**int**>>& matrix) {

**int** n = matrix.size();

**if**(n==0) **return**;

rotate(matrix, 0, n-1);

}

**void** rotate(vector<vector<**int**>>& matrix, **int** x, **int** n){

**int** i;

**if**(x==n) **return**;

**for**(i=0;i<n-x;i++) { // i = distance from x

**int** tmp = matrix[x][x+i];

swap(tmp, matrix[x+i][n]);

swap(tmp, matrix[n][n-i]);

swap(tmp, matrix[n-i][x]);

swap(tmp, matrix[x][i+x]);

}

x++; n--;

**if**(n >= x) rotate(matrix,x,n);

}

};

# String ----Trie

## ►Implement Trie (Prefix Tree)

Implement a trie with insert, search, and startsWith methods.

**Note:**  
You may assume that all inputs are consist of lowercase letters a-z.

**struct** TrieNode{

**char** content;

**bool** wordEnd = **false**;

TrieNode\* children[26];

};

**class** Trie {

**public**:

Trie() {

root = **new** TrieNode();

}

// Inserts a word into the trie.

**void** insert(string word)

{

TrieNode\* p = root;

TrieNode\* tmp;

**for** (**int** i = 0; i < word.length(); ++i) {

tmp = p->child[word[i] - 'a'];

**if** (tmp == NULL) {

p->children[word[i] - 'a'] = **new** TrieNode();

}

p = p->children[word[i] - 'a'];

}

p->wordEnd = **true**;

}

// Returns if the word is in the trie.

**bool** search(string word)

{

TrieNode\* p = root;

TrieNode\* tmp;

**int** i;

**for** (i = 0; i < word.length(); ++i) {

tmp = p->children[word[i] - 'a'];

**if** (tmp == NULL)

**break**;

p = tmp;

}

**if**(i == word.length() && p->wordEnd)

**return** **true**;

**else**

**return** **false**;

}

// Returns if there is any word in the trie

// that starts with the given prefix.

**bool** startsWith(string prefix)

{

TrieNode\* p = root;

TrieNode\* tmp;

**for** (**int** i = 0; i < prefix.length(); ++i){

tmp = p->children[prefix[i] - 'a'];

**if** (tmp == NULL) **return** **false**;

p = tmp;

}

**return** **true**;

}

**private**:

TrieNode\* root;

};

## ►Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

void addWord(word)

bool search(word)

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

For example:

addWord("bad")

addWord("dad")

addWord("mad")

search("pad") -> false

search("bad") -> true

search(".ad") -> true

search("b..") -> true

**Note:**  
You may assume that all words are consist of lowercase letters a-z.

#include <string>

#include <iostream>

#include <vector>

#include <fstream>

**using** **namespace** std;

**static** **const** **int** N = 26;

**class** TrieNode {

**public**:

**bool** wordEnd;

**char** c;

TrieNode\* children[N];

TrieNode() {

wordEnd = **false**;

**for**(**int** i=0;i<N;i++) children[i] = NULL;

}

};

**class** Trie {

**public**:

Trie() {

root = **new** TrieNode();

}

// Adds a word into the data structure.

**void** insert(string word) {

TrieNode\* p = root;

**for** (**int** i=0; i<word.length(); i++) {

**if** (!(p->children[word[i] - 'a'])) {

TrieNode\* tmp = **new** TrieNode();

p->children[word[i] - 'a'] = tmp;

tmp->c = word[i];

}

p = p->children[word[i] - 'a'];

}

p->wordEnd = **true**;

}

// Returns if the word is in the Trie. A word could

// contain the dot character '.' to represent any one letter.

**bool** find(string word) {

**return** findHelper(word.c\_str(), root);

}

**void** dumpAlphabetically()

{

cout<<"Print all words in alphabetical order:"<<endl;

vector<**char**> path;

dumpAlphabeticallyHelper(root, path);

}

**private**:

TrieNode\* root;

**bool** findHelper(**const** **char**\* word, TrieNode\* node) {

TrieNode\* p = node;

**for** (**int** i = 0; word[i]; i++) {

**if** (p && word[i] != '.'){

p = p->children[word[i] - 'a'];

} **else** **if** (p && word[i] == '.') {

TrieNode\* tmp = p;

**for** (**int** j = 0; j < N; j++) {

p = tmp -> children[j];

**if** (findHelper(word + i + 1, p))

**return** **true**;

}

} **else** **break**;

}

**return** p && p->wordEnd;

}

**void** dumpAlphabeticallyHelper(TrieNode \*root, vector<**char**> &path)

{

**int** i;

TrieNode\* p;

**if**(root == NULL) **return**;

**if**(root->wordEnd) {

string s;

**for**(i=0;i<path.size();i++) s += path[i];

cout<<s<<endl;

}

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

path.push\_back('a'+i);

dumpAlphabeticallyHelper(root->children[i], path);

path.pop\_back();

}

}

};

**int** main(**int** argc, **char**\* argv[])

{

**if** (argc < 2) {

cerr << "Usage: " << argv[0] << " FileName" <<endl;

**return** 1;

}

Trie w;

string word;

ifstream infile(argv[1]);

cout<<"Read word from file and insert to Trie:"<<endl;

**if**( infile.is\_open() ){

**while**( infile >> word){

cout<<"["<<word<<"]"<<endl;

w.insert(word);

}

}**else**{

cout<<"Unable to open file"<<endl;

**return** 1;

}

w.dumpAlphabetically();

cout<<endl;

cout<<"w.find(\".\") -> "<<w.find(".")<<endl;

cout<<"w.find(\"aa\") -> "<<w.find("aa")<<endl;

cout<<"w.find(\"a\") -> "<<w.find("a")<<endl;

cout<<"w.find(\".a\") -> "<<w.find(".a")<<endl;

cout<<"w.find(\"..t\") -> "<<w.find("..t")<<endl;

cout<<"w.find(\"car\") -> "<<w.find("car")<<endl;

cout<<"w.find(\"c.t\") -> "<<w.find("c.t")<<endl;

**return** 0;

}

# String

## Last Character of a encoded string

#include <iostream>

**using** **namespace** std;

// To execute C++, please define "int main()"

/\*A string is encoded as a list of bytes (integers from 0 to 255) in the following way:

- If a byte starts with a 0 (from 0b0000 0000 to 0b0111 1111, or 0 to 127, or 0x00 to 0x7F), the character value is that byte.

- If a byte starts with a 1 (from 0b1000 0000 to 0b1111 1111, or 128 to 255, or 0x80 to 0xFF), the character value is that byte, followed by the next byte, with the most significant bit dropped. For example, 0x81 followed by 0x7F has a character value of 0x017F.

Note that there is thus a difference between a byte and a character. In particular, a character might be either 1 or 2 bytes in length.

Find the character value of the last character in a given string.

For example:

Input: [0x03, 0xFF, 0xFF, 0x7F]

Output: 1

Input: [0xFF, 0x03 , 0xFF, 0x7F]

Output: 2

input [1,2, 2,1,1]

 1, odd 2s 1 last one is 2 bytes charactor

 1, even 2s 1 last one is 1 bytes charactor

\*/

**bool** isTow(**unsigned** **char** c)

{

**if**(c>>7 == 1) **return** **true**;

**else** **return** **false**;

}

**int** lastCh(**unsigned** **char** s[], **int** n)

{

**int** i, count = 0;

**if**(n<2) **return** 1;

**for**(i=n-2;i>=0;i--){

**if**(isTow(s[i])){

count ++;

}**else**{

**if**(count!=0) **return** count%2==0? 1:2;

}

}

**return** count%2==0? 1:2;

}

**int** main() {

**unsigned** **char** s[] = {0x03, 0xFF, 0xFF, 0x7F};

cout<<lastCh(s, **sizeof**(s))<<endl;

**unsigned** **char** s1[] = {0xFF ,0x03, 0xFF, 0x7F};

cout<<lastCh(s1, **sizeof**(s1))<<endl;

**unsigned** **char** s2[] = {0xFF ,0x03 };

cout<<lastCh(s2, **sizeof**(s2))<<endl;

**return** 0;

}

## ►►►Wildcard Matching

Implement wildcard pattern matching with support for '?' and '\*'.

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "\*") → true

isMatch("aa", "a\*") → true

isMatch("ab", "?\*") → true

isMatch("aab", "c\*a\*b") → false

Method 1: Recursive

Space: N(1) Running Time :O(2^2)

**class** Solution {

**public**:

**bool** isMatch(string &s, string &p,**int** i, **int** j) {

**if**(p.length()==j) **return** s.length()==i;

**if**(p[j]=='\*'){

**while**(j<p.length() && p[j]=='\*') j++;

**while**(i<s.length()){

**if**(isMatch(s,p,i,j)) **return** **true**;

i++;

}

**return** isMatch(s,p,i,j);

}**else** **if**(i<s.length() && (p[j]=='?' || s[i]==p[j])){

**return** isMatch(s,p,i+1,j+1);

}**else** **return** **false**;

}

};

Method 2: DP

Space: N(nm) Running Time :O(nm) 296 ms

**class** Solution {

**public**:

**bool** isMatch(string s, string p) {

**int** pl = 0;

**for** (**int** i = 0; i < p.length(); i++) {

**if** (p[i] != '\*') pl++;

}

**if** (pl > s.length()) **return** **false**;

**if** (s.length() == 0 && pl == 0) **return** **true**;

vector<vector<**bool**>> dp(s.length()+1,

vector<**bool**>(p.length()+1));

dp[0][0] = **true**;

**for** (**int** i = 1; i <= p.length(); i++) {

**if**(p[i-1] == '\*') {

dp[0][i] = dp[0][i-1];

}**else**

dp[0][i] = **false**;

}

**for** (**int** i=1; i<=s.length(); i++) {

**for** (**int** j = 1; j <= p.length(); j++) {

**if** (p[j-1] == '?'|| p[j-1] == s[i-1]) {

dp[i][j] = dp[i-1][j-1];

} **else** **if** (p[j-1] == '\*') {

dp[i][j] = dp[i-1][j] || dp[i][j-1];

} **else** {

dp[i][j] = **false**;

}

}

}

**return** dp[s.length()][p.length()];

}

};

Method 3: DP reduce the space

Space: O(n) Runing Time: O(nm) 430ms

**class** Solution {

**public**:

**bool** isMatch(string s, string p) {

**int** pl = 0;

**for** (**int** i = 0; i < p.length(); i++) {

**if** (p[i] != '\*') pl++;

}

**if** (pl > s.length()) **return** **false**;

**if** (s.length() == 0 && pl == 0) **return** **true**;

// main function

vector<vector<**bool**>> dp(2,

vector<**bool**>(s.length()+1));

dp[0][0] = **true**;

dp[1][0] = **false**;

**for** (**int** ps = 1; ps <= p.length(); ps++) {

// remember to check the first one.

// make "\*" can match ""

**if** (p[ps-1] == '\*') {

dp[1][0] = dp[0][0];

} **else** {

dp[1][0] = **false**;

}

**for** (**int** ss = 1; ss <= s.length(); ss++) {

**if** (p[ps-1] == '?' || p[ps-1] == s[ss-1]) {

dp[1][ss] = dp[0][ss-1];

} **else** **if** (p[ps-1] == '\*') {

dp[1][ss] = dp[1][ss - 1] || dp[0][ss];

} **else** {

dp[1][ss] = **false**;

}

}

**for** (**int** i = 0; i <= s.length(); i++) {

dp[0][i] = dp[1][i];

}

}

**return** dp[1][s.length()];

}

};

Method 4: linear runtime and constant space solution

Space: O(1) Time: O(n+m) 8ms

**bool** isMatch(**const** **char** \*s, **const** **char** \*p) {

**const** **char**\* star=NULL;

**const** **char**\* ss=s;

**while** (\*s){

**if** ((\*p=='?')||(\*p==\*s)){

s++;p++;

}**else** **if** (\*p=='\*'){

star=p;

p++;

ss=s;

}**else** **if** (star){

p = star+1;

ss++;

s=ss;

}**else** **return** **false**;

}

**while** (\*p=='\*') p++;

**return** !\*p;

}

## ►Longest Palindromic Substring

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

class Solution {

public:

string longestPalindrome(string s) {

if (s.empty()) return "";

if (s.size() == 1) return s;

int min\_start = 0, max\_len = 1;

for (int i = 0; i < s.size();) {

if (s.size() - i <= max\_len / 2) break;

int j = i, k = i;

while (k < s.size()-1 && s[k+1] == s[k]) ++k; // Skip duplicate characters.

i = k+1;

while (k < s.size()-1 && j > 0 && s[k + 1] == s[j - 1]) { ++k; --j; } // Expand.

int new\_len = k - j + 1;

if (new\_len > max\_len) { min\_start = j; max\_len = new\_len; }

}

return s.substr(min\_start, max\_len);

}

};

## Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, non-negative integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

"1 + 1" = 2

" 2-1 + 2 " = 3

"(1+(4+5+2)-3)+(6+8)" = 23

class Solution {

public:

int calculate(string s) {

int total = 0;

vector<int> signs(2, 1);

for (int i=0; i<s.size(); i++) {

char c = s[i];

if (c >= '0') {

int number = 0;

while (i < s.size() && s[i] >= '0') number = 10 \* number + s[i++] - '0';

total += signs.back() \* number;

signs.pop\_back();

i--;

} else if (c == ')'){

signs.pop\_back();

} else if (c != ' '){

signs.push\_back(signs.back() \* (c == '-' ? -1 : 1));

}

}

return total;

}

};

## ►Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, +, -, \*, / operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

"3+2\*2" = 7

" 3/2 " = 1

" 3+5 / 2 " = 5

class Solution {

public:

int calculate(string s) {

int len = s.length();

if(len==0) return 0;

vector<int> Stack;

int num = 0;

char sign = '+';

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

if(isdigit(s[i])){

num = num\*10+s[i]-'0';

}

if((!isdigit(s[i]) &&' '!=s[i]) || i==len-1){

if(sign=='-'){

Stack.push\_back(-num);

}

if(sign=='+'){

Stack.push\_back(num);

}

if(sign=='\*'){

int tmp = Stack.back()\*num;

Stack.pop\_back();

Stack.push\_back(tmp);

}

if(sign=='/'){

int tmp = Stack.back()/num;

Stack.pop\_back();

Stack.push\_back(tmp);

}

sign = s[i];

num = 0;

}

}

int re = 0;

for(auto i:Stack){

re += i;

}

return re;

}

};

## Evaluate Reverse Polish Notation

Question Solution

Evaluate the value of an arithmetic expression in [Reverse Polish Notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation).

Valid operators are +, -, \*, /. Each operand may be an integer or another expression.

Some examples:

["2", "1", "+", "3", "\*"] -> ((2 + 1) \* 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

**Learn Something:**

1. Convert c style string to int, first check string is “0”?

#include <stdlib.h> /\* atoi \*/

int atoi (const char \* str);

return 0: faild

Runtime: **56 ms**

class Solution {

public:

int evalRPN(vector<string> &tokens) {

string t;

int a,b,num;

vector<int> stack;

for(int i=0; i<tokens.size(); i++){

t = tokens[i];

num = 1;

num = atoi(t.c\_str());

if(t.compare("0")==0){

num = 0;

stack.push\_back(num);

}else if( num == 0){

b = stack.back();

stack.pop\_back();

a = stack.back();

stack.pop\_back();

if(t.compare("+")==0){

stack.push\_back( a + b);

}else if(t.compare("-")==0){

stack.push\_back( a - b);

}else if(t.compare("\*")==0){

stack.push\_back( a \* b);

}else if(t.compare("/")==0){

stack.push\_back( a / b);

}

}else{

stack.push\_back(num);

}

}

return stack[0];

}

};

## Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,  
**path** = "/home/", => "/home"  
**path** = "/a/./b/../../c/", => "/c"

**Corner Cases:**

* Did you consider the case where **path** = "/../"?  
  In this case, you should return "/".
* Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/".  
  In this case, you should ignore redundant slashes and return "/home/foo".
* Input: "/..." Expected: "/..."

**class** Solution {

**public**:

string simplifyPath(string path) {

**int** i;

**int** n = path.length();

string ret;

vector<string> S;

vector<string> tmp;

i=0;

**while**(i<n){

**char** c = path[i];

string word = "";

**while**(c!='/') {

word += path[i];

i++;

**if**(i>=n) **break**;

c = path[i];

}

**if**(word.compare("")!=0){

S.push\_back(word);

**continue**;

}

i++;

}

**for**(string d:S){

**if**(d.compare("..")==0){

**if**(!tmp.empty()) tmp.pop\_back();

}**else** **if**(d.compare(".")==0) ;

**else** tmp.push\_back(d);

}

**for**(string d:tmp){

ret += "/"+d;

}

**if**(ret.compare("")==0) ret = "/";

**return** ret;

}

};

## ►►►String ----Rolling Hash

// ASCII a = 97, b = 98, r = 114.

hash("abr") = (97 × 101^2) + (98 × 101^1) + (114 × 101^0) = 999,509

// base old hash old 'a' new 'a'

hash("bra") = [101 × (999,509 - (97 × 101^2))] + (97 × 101^0) = 1,011,309

http://en.wikipedia.org/wiki/Rabin%E2%80%93Karp\_algorithm

[**C++ code of the Rabin Karp Algo given in the CLRS book(7ms)**](https://leetcode.com/discuss/24391/c-code-of-the-rabin-karp-algo-given-in-the-clrs-book-7ms)

**class** Solution {

**public**:

**int** strStr(string haystack, string needle) {

// to find the big prime number: http://www.bigprimes.net/archive/prime/100001/

**unsigned** **int** q = 373591679;

**int** m = needle.length();

**int** n = haystack.length();

**if** (m == 0 || n == 0 || m > n)

**return** m == 0 ? 0 : -1;

**int** d = 3;

**int** t = haystack[0];

**int** p = needle[0];

**int** h = 1;

**for**(**int** i = 1; i < m; i++) {

h = (h\*d)%q;

p = (p\*d + needle[i])%q;

t = (t\*d + haystack[i])%q;

}

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

**if**(p == t) {

**int** j = 0;

**for**(; j < m && needle[j] ==haystack[i+j]; j++);

**if**(j == m)

**return** i;

}

**if**(i < n-m)

t = ((t-haystack[i]\*h)\*d + haystack[i+m])%q;

}

**return** -1;

}

};

## ►►►Text Justification

 Given an array of words and a length *L*, format the text such that each line has exactly *L* characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly*L* characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,  
**words**: ["This", "is", "an", "example", "of", "text", "justification."]  
**L**: 16.

Return the formatted lines as:

[

"This is an",

"example of text",

"justification. "

]

**Note:** Each word is guaranteed not to exceed *L* in length.

**Corner Cases:**

* A line other than the last line might contain only one word. What should you do in this case?  
  In this case, that line should be left-justified.

**Solution:** For each line, I first figure out which words can fit in. According to the code, these words are words[i] through words[i+k-1]. Then spaces are added between the words. The trick here is to use mod operation to manage the spaces that can't be evenly distrubuted: the first (L-l) % (k-1) gaps acquire an additional space.

**class** Solution {

**public**:

vector<string> fullJustify(vector<string> &words, **int** L) {

vector<string> res;

**for**(**int** i = 0, k, l; i < words.size(); i += k) {

**for**(k = l = 0; i + k < words.size() **and** l + words[i+k].size() <= L - k; k++) {

l += words[i+k].size();

}

string tmp = words[i];

**for**(**int** j = 0; j < k - 1; j++) {

**if**(i + k >= words.size()) tmp += " ";

**else** tmp += string((L - l) / (k - 1) + (j < (L - l) % (k - 1)), ' ');

tmp += words[i+j+1];

}

tmp += string(L - tmp.size(), ' ');

res.push\_back(tmp);

}

**return** res;

}

};

# Heap

## Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,  
Given [3,2,1,5,6,4] and k = 2, return 5.

Note: You may assume k is always valid, 1 ≤ k ≤ array's length.

**class** Solution {

**public**:

**int** findKthLargest(vector<**int**>& nums, **int** k) {

**int** i;

**int** n = nums.size();

priority\_queue<**int**,vector<**int**>, greater<**int**> > q;

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

q.push(nums[i]);

}

**for**(i=k;i<n;i++){

**if**(nums[i]>q.top()){

q.pop();

q.push(nums[i]);

}

}

**return** q.top();

}

};

# Hash

## ►Anagrams

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

First I used this hash function, that make every string has the same sum has the same hash value.

**long** hash(string &s){

**long** h = 0;

**for**(**int** i=0;i<s.length();i++){

h += (s[i]-'a')\*3571 + 3559;

}

**return** h;

}

Input:**["cab","tin","pew","duh","may","ill","buy","bar","max","doc"]**

Output:**["duh","ill"]**

Expected:**[]**

**class** Solution {

**public**:

vector<string> anagrams(vector<string>& strs) {

vector<string> res;

unordered\_map<**long**, vector<**int**>> m;

unordered\_set<string> s;

**int** i;

**for**(i=0;i<strs.size();i++){

**long** h = hash(strs[i]);

**if**(m.count(h)>0){

m[h].push\_back(i);

}**else**{

m[h] = vector<**int**>(1,i);

}

}

**for** (**auto**& kv : m) {

vector<**int**> &p = kv.second;

**if**(p.size()>1)

**for**(i=0;i<p.size();i++)

res.push\_back( strs[p[i]] );

}

**return** res;

}

**long** hash(string &s){

**long** h = 0;

**for**(**int** i=0;i<s.length();i++){

h += s[i]\*s[i]\*s[i]\*s[i]\*3571 ;

}

**return** h;

}

};

Another Hash Function

**long** hash(string &str){

**static** **int** PRIMES[] = {2, 3, 5, 7, 11 ,13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 107};

**long** h = 1;

**for**(**int** i=0;i<str.length();i++){

h \*= PRIMES[str[i]-'a'] ;

}

**return** h;

}

## ►Longest Sequence in Array

Given an int array which might contain duplicates, find the largest subset of it which form a sequence.

Eg. {1,6,10,4,7,9,5}

then ans is 4,5,6,7

Sorting is an obvious solution. Can this be done in O(n) time

O(n) time and space

In Python, dict is a hash with O(1) time to retrieve an item.

okay here is an explanation:

Logic here is that each element in the hashtable keeps track of the sequence. So boundary elements of the sequence are important as the end element of the sequence points to beginning element of the sequence and beginning element of the sequence points to the end element. So whenever a new element is to be added to the sequence it picks up the boundary value and becomes the new boundary. Check how 8 is added to the table below:

Array a: [1, 6, 10, 4, 7, 8]

i: 1 Table: {1=1}

i: 6 Table: {1=1, 6=6}

i: 10 Table: {1=1, 6=6, 10=10}

i: 4 Table: {1=1, 4=4, 6=6, 10=10}

i: 7 Table: {1=1, 4=4, 6=7, 7=6, 10=10}

i: 8 Table: {1=1, 4=4, 6=8, 7=6, 8=6, 10=10}

def find(arr):

table = {}

first = 0

last = 0

for i in arr:

beg = end = i

if i in table:

continue

table[i] = 'EXISTED'

if i - 1 in table:

beg = table[i-1]

if i + 1 in table:

end = table[i+1]

table[beg] = end

table[end] = beg

if end - beg > last - first:

first = beg

last = end

return list(range(first, last + 1))

arr = [1,6,10,4,7,9,5, 5,8]

print(find(arr))

## ►Minimum Interval in N Sorted List

You have k lists of sorted integers. Find the smallest range that includes at least one number from each of the k lists.

For example,

List 1: [4, 10, 15, 24, 26]

List 2: [0, 9, 12, 20]

List 3: [5, 18, 22, 30]

The smallest range here would be [20, 24] as it contains 24 from list 1, 20 from list 2, and 22 from list 3.

Solution: There are k lists of sorted integers. Make a min heap of size k containing 1 element from each list. Keep track of min and max element and calculate the range.   
In min heap, minimum element is at top. Delete the minimum element and another element instead of that from the same list to which minimum element belong. Repeat the process till any one of the k list gets empty.   
Keep track of minimum range.   
  
For eg.   
List 1: [4, 10, 15, 24, 26]   
List 2: [0, 9, 12, 20]   
List 3: [5, 18, 22, 30]   
  
Min heap of size 3. containing 1 element of each list   
Heap [0, 4, 5]   
Range - 6   
  
Remove 0 and add 9   
Heap [4, 9, 5]   
Range - 6   
  
Remove 4 and add 10   
Heap [5, 9, 10]   
Range - 6   
  
and so on....   
  
Finally you will yield the result. When you are creating the heap for the first time, make a separate variable that keeps track of max number in the heap. Everytime you add a new element into the heap, check the new element against this variable. That way you can get the max-min range

**Correctness:** This is the same as “merge all the N lists into one long array, then keep a sliding window of N”.

from:2 1 3 2 1 2 1 3 3 2 1 1 3

Num:0 4 5 9 10 12 15 18 22 20 24 26 30

struct pn

{

int n; /\* belong to which array \*/

int d; /\* the data value \*/

pn(int \_n, int \_d) { n = \_n; d = \_d; }

pn(const pn& \_pn) { n = \_pn.n; d = \_pn.d; }

};

inline void swap(pn& a, pn& b) { pn c = a; a = b; b = c; }

void adjust(int n, pn a[])

{

int i = 0, max = 0;

int l = 0, r = 0;

for(i = n / 2; i >= 0; i--)

{

max = i;

l = 2 \* i + 1;

r = 2 \* i + 2;

if(l < n && a[l].d > a[max].d) { max = l; }

if(r < n && a[r].d > a[max].d) { max = r; }

if(max != i) { swap(a[max], a[i]); }

}

}

void heapsort(int n, pn a[])

{

int i = 0;

adjust(n, a);

for(i = n - 1; i > 0; i--)

{

swap(a[0], a[i]);

adjust(i, a);

}

}

int main()

{

int i = 0, j = 0;

const int m = 3;

const int n = 5;

int ms = 0, me = 0;

int ts = 0, te = 0;

int a[m][n] = { {4, 10, 15, 24, 26}, {0, 9, 12, 20, 35}, {5, 18, 22, 30, 50} };

int cur[m] = {1, 1, 1}; /\* record the current positions of each array which haven't been used \*/

pn heap[m] = {pn(0, a[0][0]), pn(1, a[1][0]), pn(2, a[2][0])};

heapsort(m, heap);

ms = heap[0].d;

me = heap[m - 1].d;

while(true)

{

heapsort(m, heap);

ts = heap[0].d;

te = heap[m - 1].d;

/\* if the current range is smaller than the minimum range \*/

if(te - ts < me - ms) { ms = ts; me = te; }

/\* if the sub-array which the smallest element comes from hasn't to the end \*/

if(cur[heap[0].n] != n)

{

heap[0].d = a[heap[0].n][cur[heap[0].n]];

cur[heap[0].n] += 1;

}

else

{

break;

}

}

cout << ms << endl;

cout << me << endl;

return 0;

}

## ►►►Substring with Concatenation of All Words

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words**exactly once and without any intervening characters.

For example, given:  
**s**: "barfoothefoobarman"  
**words**: ["foo", "bar"]

You should return the indices: [0,9].  
(order does not matter).

Failed Cases:

Input:**"barfoothefoobarman", ["foo","bar"]**

Output:**[0,6,9]**

Expected:**[0,9]**

Input:**"wordgoodgoodgoodbestword", ["word","good","best","good"]**

Output:**[4,8]**

Expected:**[8]**

**Solution 1: Naive Solution using two unordered\_map**

We use an unordered\_map<string, int> counts to record the expected times of each word and another unordered\_map<string, int> seen to record the times we have seen. Then we check for every possible position of i. Once we meet an unexpected word or the times of some word is larger than its expected times, we stop the check. If we finish the check successfully, push i to the result indexes.

vector<**int**> findSubstring(string s, vector<string>& words) {

unordered\_map<string, **int**> counts;

**for** (string word : words) counts[word]++;

**int** n = s.length();

**int** m = words.size();

**int** len = words[0].length();

vector<**int**> res;

**for** (**int** i = 0; i < n - m \* len + 1; i++) {

unordered\_map<string, **int**> seen;

**int** j = 0;

**for** (; j < m; j++) {

string word = s.substr(i + j \* len, len);

**if** (counts.count(word)>0) {

seen[word]++;

**if** (seen[word] > counts[word])

**break**;

}**else** **break**;

}

**if** (j == m) res.push\_back(i);

}

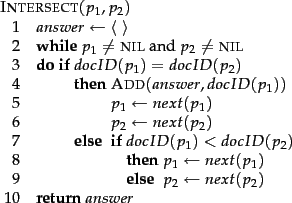
**return** res;

}

# English

## Describe Merge Sorted List Algorithm

We maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers.



# Facebook 面经

## isDistanceZeroOrOne

class IntFileIterator {  
boolean hasNext();  
int next();  
}

class{  
public boolean isDistanceZeroOrOne(IntFileIterator a, IntFileIterator b)；

}  
// return if the distance between a and b is at most 1..  
// Distance: minimum number of modifications to make a=b  
// Modification:  
// 1. change an int in a  
// 2. insert an int to a  
// 3. remove an int from a

这题就是one edit distance的变形题，难点在于给的Iterator，事先不知道两个file的长度，也不允许用extra space（所以不能转成两个string再比），那么一个个往前跑的时候就得三种情况都考虑。。。。

难在不能回头，所以用复制两个新的，然后递归。

**public** **bool** isDistanceZeroOrOne(ref **int** cnt,

IntFileIterator A,

IntFileIterator B)

{

**while** (A.hasNext() && B.hasNext()) {

IntFileIterator copyA = A;

IntFileIterator copyB = B;

**int** a = A.next();

**int** b = B.next();

**if** (a == b) **continue**;

**else** {

cnt++;

**if**(cnt >= 2) **return** **false**;

**return** isDistanceZeroOrOne(ref cnt, copyA, B) ||

isDistanceZeroOrOne(ref cnt, A, copyB) ||

isDistanceZeroOrOne(ref cnt, copyA, copyB);

}

}

**while**(A.hasNext()) cnt++;

**while**(B.hasNext()) cnt++;

**return** cnt >= 2 ? **false**: **true**;

}

# Google 面经

## Next Prime Number

Given an integer, return the next prime number bigger than it.  
Ans:

[http://stackoverflow.com/questions/2468412/finding-a-prime-numb](http://stackoverflow.com/questions/2468412/finding-a-prime-number-after-a-given-number)

## New Random()

Given random() that can return 0 or 1 uniformly, implement random\_new() that can return 0 with 90%, and 1 with 10%.

Solution: 算10次，2进制是0-1023，0-99 return 1;100-999 return 0;其他再算一次.

运行四次就行了吧。0000~1111  
0000到1001的话返回0,1010返回1，其它调用自身.缺点是调用自身的概率会有点大，优点就是不用运行那么多次原函数.不过也不好说，都是tradeoff  
  
Design a distributed LRU  
  
Design

## suppose you have a cluster, and each machine in this cluster has a large number of numbers. How can you find out the median of all the numbers on all the machines. Ring

Given a set of points on a plane, and a list of circles centered at the original point, find the ring containing the most number of points.

# AirBNB 面经

## ►CoinChange.java

import java.util.\*;

/\*\*

\* Return the coins combination with the minimum number of coins.

\* Time complexity O(MN), where M is the target value and N is the number of distinct coins.

\* Space complexity O(M).

\*/

public class CoinChange {

public int[] minCoins(int value, int[] coins) {

Arrays.sort(coins);

int[] cache = new int[value + 1]; // stores the min number of coins

Arrays.fill(cache, Integer.MAX\_VALUE);

int[] prev = new int[value + 1]; // stores the previous location

cache[0] = 0;

prev[0] = 0;

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

for (int j = coins.length - 1; j >= 0; --j) {

if (coins[j] <= i) {

if (1 + cache[i - coins[j]] < cache[i]) { // find better combination

cache[i] = 1 + cache[i - coins[j]];

prev[i] = i - coins[j];

}

}

}

}

int[] res = new int[cache[value]];

int idx = 0;

int remain = value;

while (remain != 0) {

int coin = remain - prev[remain];

remain = prev[remain];

res[idx] = coin;

++idx;

}

return res;

}

}

## ►LowerUpperCasePermutation.java

import java.util.\*;

/\*\*

\* Given a string, generate all strings with identical character sequence,

\* but with different upper/lower cases.

\*

\* Time complexity: O(2^N), Space complexity: O(2^N).

\*/

public class LowerUpperCasePermutation {

public List<String> lowerUpperCasePermutation(String str) {

List<String> res = new ArrayList<String>();

if (str.length() == 0) {

return res;

}

String upperCase = str.toUpperCase();

int pos = 0;

char[] chars = upperCase.toCharArray();

generate(chars, pos, res);

return res;

}

private void generate(char[] chars, int pos, List<String> res) {

if (pos == chars.length) {

res.add(new String(chars));

} else {

// current character is upper case

generate(chars, pos + 1, res);

// current character is lower case

chars[pos] = Character.toLowerCase(chars[pos]);

generate(chars, pos + 1, res);

chars[pos] = Character.toUpperCase(chars[pos]);

}

}

}

## ►KEditDistance.java

import java.util.\*;

public class KEditDistance {

public Set<String> kDistance(String target, List<String> words, int k) {

Set<String> res = new HashSet<String>();

for (String candidate : words) {

if (similar(target, candidate, k)) {

res.add(candidate);

}

}

return res;

}

private boolean similar(String target, String candidate, int k) {

int first = target.length();

int second = candidate.length();

int[][] dist = new int[first + 1][second + 1];

for (int r = 0; r <= first; ++r) {

for (int c = 0; c <= second; ++c) {

if (r == 0 && c == 0) {

dist[r][c] = 0;

} else if (r == 0) {

dist[r][c] = c;

} else if (c == 0) {

dist[r][c] = r;

} else {

if (target.charAt(r - 1) == candidate.charAt(c - 1)) {

dist[r][c] = dist[r - 1][c - 1];

} else {

dist[r][c] = Math.min(dist[r - 1][c - 1], Math.min(dist[r][c - 1], dist[r - 1][c])) + 1;

}

if (r == c && dist[r][c] > k) {

return false;

}

}

}

}

return true;

}

}

## ►ParseCSV.java

import java.util.\*;

/\*\*

\* Parse the csv string, considering the escapes.

\*

\* Time complexity: O(N), space complexity: O(N).

\*

\*/

public class ParseCSV {

public String[] parseCSV(String line) {

boolean prevEscape = false;

List<String> list = new ArrayList<String>();

StringBuilder sb = new StringBuilder();

for (int i = 0; i < line.length(); ++i) {

char cur = line.charAt(i);

if (cur == '\\' && !prevEscape) {

prevEscape = true;

} else if (cur == ',') {

if (prevEscape) {

sb.append(',');

prevEscape = false;

} else {

list.add(sb.toString().trim());

sb = new StringBuilder();

}

} else {

sb.append(cur);

prevEscape = false;

}

}

list.add(sb.toString().trim());

String[] res = new String[list.size()];

list.toArray(res);

return res;

}

}

## ►Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

if(n==m) return n;

int dif = n - m;

int b = 1;

int sum = 0;

while(b>0){

if(dif < b && n&b && m&b) sum += b;

b = b<<1;

}

return sum;

}

};

# 

# Unclassified

## ►►Insert Interval

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

**Example 1:**  
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

**Example 2:**  
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

**Solution:**

By far the best solution I have seen is of O(n) time (some solutions claim to be of O(logn) turns out to be O(n)). One of the simplest ideas is to compare each interval in ntervals(intervals[i]) with newInterval and then perform respective operations according to their relationships.

1. If they overlap, merge them to newInterval;
2. If intervals[i] is to the left of newInterval, push intervals[i] to the result vector;
3. If newInterval is to the left of intervals[i], push newInterval and all the remaining intervals (intervals[i], ..., intervals[n - 1]) to the result vector.

The code is as follows.

/\*\*

 \* Definition for an interval.

 \* struct Interval {

 \* int start;

 \* int end;

 \* Interval() : start(0), end(0) {}

 \* Interval(int s, int e) : start(s), end(e) {}

 \* };

 \*/

**class** Solution {

**public**:

vector<Interval> insert(vector<Interval>& intervals,

Interval newI) {

vector<Interval> res;

**int** n = intervals.size();

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

**if** (intervals[i].end < newI.start)

res.push\_back(intervals[i]);

**else** **if** (newI.end < intervals[i].start) {

res.push\_back(newI);

**for** (**int** j = i; j < n; j++)

res.push\_back(intervals[j]);

**return** res;

}

**else** newI = merge(intervals[i], newI);

}

res.push\_back(newI);

**return** res;

}

**private**:

Interval merge(Interval interval1, Interval interval2) {

**int** start = min(interval1.start, interval2.start);

**int** end = max(interval1.end, interval2.end);

**return** Interval(start, end);

}

};

## ►►►First Missing Positive

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.

**Solution:**

Since we can not use extra space, so thinking about using the nums vector itself to record a positive number occurred.

First, put each number in its right place.

For example:

When we find 5, then swap it with A[4].

At last, the first place where its number is not right, return the place + 1.

**class** Solution {

**public**:

**int** firstMissingPositive(vector<**int**>& nums) {

**for**(**int** i=0; i<nums.size(); i++){

**if**(i+1==nums[i]) **continue**;

**int** x = nums[i];

**while**(x>=1 && x<=nums.size() && x!=nums[x-1]){

swap(x, nums[x-1]);

}

}

**for**(**int** i=0; i<nums.size(); i++){

**if**(i+1!=nums[i]) **return** i+1;

}

**return** nums.size()+1;

}

};

# Reservoir Sampling

[Reservoir sampling](http://en.wikipedia.org/wiki/Reservoir_sampling) is a family of randomized algorithms for randomly choosing *k*samples from a list of *n* items,where *n* is either a very large or unknown number. Typically *n*is large enough that the list doesn’t fit into main memory. For example, a list of search queries in Google and Facebook.

So we are given a big array (or stream) of numbers (to simplify), and we need to write an efficient function to randomly select *k* numbers where *1 <= k <= n*. Let the input array be *stream*[ ]*.*

A **simple solution**is to create an array *reservoir[]* of maximum size *k*. One by one randomly select an item from*stream[0..n-1]*. If the selected item is not previously selected, then put it in *reservoir[]*. To check if an item is previously selected or not, we need to search the item in *reservoir[]*. The time complexity of this algorithm will be*O(k^2)*. This can be costly if *k* is big. Also, this is not efficient if the input is in the form of a stream.

It **can be solved in *O(n)* time**. The solution also suits well for input in the form of stream. The idea is similar to[this](http://www.geeksforgeeks.org/archives/25111)post. Following are the steps.

**1)** Create an array *reservoir[0..k-1]* and copy first *k* items of *stream[]* to it.  
**2)**Now one by one consider all items from *(k+1)*th item to *n*th item.  
…**a)** Generate a random number from 0 to *i* where *i* is index of current item in *stream*[ ]. Let the generated random number is *j*.  
…**b)** If*j* is in range 0 to *k-1*, replace *reservoir[j]* with *arr[i]*

Following is C implementation of the above algorithm.

// An efficient program to randomly select k items from a stream of items

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// A utility function to print an array

**void** printArray(**int** stream[], **int** n)

{

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

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

printf("\n");

}

// A function to randomly select k items from stream[0..n-1].

**void** selectKItems(**int** stream[], **int** n, **int** k)

{

**int** i; // index for elements in stream[]

// reservoir[] is the output array. Initialize it with

// first k elements from stream[]

**int** reservoir[k];

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

reservoir[i] = stream[i];

// Use a different seed value so that we don't get

// same result each time we run this program

srand(time(NULL));

// Iterate from the (k+1)th element to nth element

**for** (; i < n; i++)

{

// Pick a random index from 0 to i.

**int** j = rand() % (i+1);

//If the randomly picked index is smaller than k, then replace

// the element present at the index with new element from stream

**if** (j < k)

reservoir[j] = stream[i];

}

printf("Following are k randomly selected items \n");

printArray(reservoir, k);

}

// Driver program to test above function.

**int** main()

{

**int** stream[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

**int** n = **sizeof**(stream)/**sizeof**(stream[0]);

**int** k = 5;

selectKItems(stream, n, k);

**return** 0;

}

Output:

Following are k randomly selected items

6 2 11 8 12

Time Complexity: O(n)

**How does this work?**  
To prove that this solution works perfectly, we must prove that the probability that any item *stream[i]*where *0 <= i < n*will be in final *reservoir[]* is *k/n*. Let us divide the proof in two cases as first*k* items are treated differently.

**Case 1: For last *n-k* stream items, i.e., for *stream[i]* where *k <= i < n***  
For every such stream item *stream[i]*, we pick a random index from 0 to *i* and if the picked index is one of the first*k* indexes, we replace the element at picked index with *stream[i]*

To simplify the proof, let us first consider the *last item*. The probability that the last item is in final reservoir = The probability that one of the first *k* indexes is picked for last item = *k/n*(the probability of picking one of the *k* items from a list of size*n*).

We can prove with **Induction**: To prove the probability for any element in the reservoir after ith round is k/i, we just need to come up with that after i+1 round the probability is k/(i+1).

Before round i+1, any element in reservoir is k/i. The probability that they are removed is k/(i+1) \* 1/k = 1/(i+1). (Select one random num from 1~i+1,and choose 1~k; replace one element in k ). So the probability they remain is i/(i+1) and thus the elements' overall probability of being in the reservoir after i rounds is k/i \* i/(i+1) = k/(i+1).

**Case 2: For first *k* stream items, i.e., for *stream[i]* where*0 <= i < k***  
The first *k* items are initially copied to *reservoir[]*and may be removed later in iterations for *stream[k]* to *stream[n]*.  
The probability that an item from *stream[0..k-1]* is in final array = Probability that the item is not picked when items*stream[k], stream[k+1], …. stream[n-1]* are considered = *[k/(k+1)] x [(k+1)/(k+2)] x [(k+2)/(k+3)] x … x [(n-1)/n] = k/n*

References:  
<http://en.wikipedia.org/wiki/Reservoir_sampling>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.