LeetCode Review

Yichuan Gui

2016

Contents

Chapter 1

Math

1.1 Nim Game (E)

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

```
class Solution {
public:
    bool canWinNim(int n) {
        if (n % 4 == 0)
            return false;
        else
            return true;
    }
};
```

1.2 Bulb Switcher (M)

There are n bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the ith round, you toggle every i bulb. For the nth round, you only toggle the last bulb. Find how many bulbs are on after n rounds.

Example: Given n = 3.

At first, the three bulbs are [off, off, off].
After first round, the three bulbs are [on, on, on].
After second round, the three bulbs are [on, off, on].
After third round, the three bulbs are [on, off, off].
So you should return 1, because there is only one bulb is on.

```
class Solution {
public:
    int bulbSwitch(int n) {
        int res = 1;
        while (res * res <= n) ++res;
        return res - 1;
        // Or just simply use one line command: return sqrt(n);
    }
};
```

1.3 Find the Celebrity (M)

Suppose you are at a party with n people (labeled from 0 to n - 1) and among them, there may exist one celebrity. The definition of a celebrity is that all the other n - 1people know him/her but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function bool knows(a, b) which tells you whether A knows B. Implement a function int findCelebrity(n), your function should minimize the number of calls to knows.

Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return -1.

```
class Solution {
public:
    int findCelebrity(int n) {
        int res = 0;
        for (int i = 0; i < n; ++i) {
            // if res knows i, then res must not be celebrity and i could be
               celebrity
            // update res to i and check the next i
            if (knows(res, i)) res = i;
```

```
        }
        for (int i = 0; i < n; ++i) {
            // if res knows i, or i doesn't know res,
            // then res must not be celebrity
            if (res != i && (knows(res, i) || !knows(i, res))) return -1;
        }
        return res;
    }
};

class Solution {
public:
    int findCelebrity(int n) {
        for (int i = 0, j = 0; i < n; ++i) {
            for (j = 0; j < n; ++j) {
                // if i knows j or j doesn't know i,
                // then i is not celebrity
                if (i != j && (knows(i, j) || !knows(j, i))) break;
            }
            // if i does not know any j , but all j know i
            // then i is celebrity
            if (j == n) return i;
        }
        return -1;
    }
};
```

1.4 Rectangle Area (E)

Find the total area covered by two rectilinear rectangles in a 2D plane. Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.

```
class Solution {
public:
    int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
        int areaA = (C-A) * (D-B);
        int areaB = (G-E) * (H-F);

        int left = max(A,E);
        int right = min(C,G);
        int top = min(D,H);
        int bottom = max(B,F);

        int overlap = 0;
        if (right > left && top > bottom)
```

```
            overlap = (right - left) * (top - bottom);

        return areaA + areaB - overlap;
    }
};
```

1.5 Container With Most Water (M)

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.

```
class Solution {
public:
    int maxArea(vector<int> &height) {
        int left = 0, right = height.size()-1, res = 0;
        while (left <= right) {
            res = max(res, min(height[left], height[right]) * (right - left));
            if (height[left] < height[right]) ++left;
            else --right;
        }
        return res;
    }
};
```

1.6 Factorial Trailing Zeroes (E)

Given an integer n, return the number of trailing zeroes in n!.

```
class Solution {
public:
    // All trailing 0 is from factors 5 * 2
    // In the n! operation, factors 2 is always ample.
    // So we just count how many 5 factors in all number from 1 to n.
    int trailingZeroes(int n) {
        if (n < 5)
            return 0;
        else
            return n/5 + trailingZeroes(n/5);
```

```
    }
};
```

1.7 Palindrome Number (E)

Determine whether an integer is a palindrome. Do this without extra space.

```
class Solution {
public:
    bool isPalindrome(int x) {
        if (x < 0) return false;

        int d = 1;
        while (x / d >= 10)
            d *= 10;          // get the initial divisor size

        while (x > 0) {
            int q = x / d;    // quotient as the first digit
            int r = x % 10;   // remainder as the last digit
            if (q != r) return false;
            x = x % d / 10;   // remove the first and the last digits
            d /= 100;         // reduce the divisor size
        }

        return true;
    }
};
```

1.8 Reverse Integer (E)

Reverse digits of an integer.

Example1: x = 123, return 321
Example2: x = -123, return -321

```
class Solution {
public:
    int reverse(int x) {
        if (x < INT_MIN || x > INT_MAX)
            return 0;

        long num, result = 0;
```

```
        if (x < 0)            // convert to positive if x is a negtive
            num = -x;
        else
            num = x;

        while (num != 0) {
            result = result * 10 + num % 10;
            num /= 10;
        }

        if (x < 0)            // convert result back if x is a negtive
            result = -result;

        if (result < INT_MIN || result > INT_MAX)
            return 0;
        else
            return result;
    }
};
```

1.9 Count Numbers with Unique Digits (M)

Given a non-negative integer n, count all numbers with unique digits, x, where $0 \leq x < 10^n$.

Example: Given n = 2, return 91. (The answer should be the total numbers in the range of $0 \leq x < 100$, excluding [11,22,33,44,55,66,77,88,99])

Hint:
A direct way is to use the backtracking approach.
Backtracking should contains three states which are (the current number, number of steps to get that number and a bitmask which represent which number is marked as visited so far in the current number). Start with state (0,0,0) and count all valid number till we reach number of steps equals to 10n.
This problem can also be solved using a dynamic programming approach and some knowledge of combinatorics.
Let f(k) = count of numbers with unique digits with length equals k.
f(1) = 10, ..., f(k) = 9 * 9 * 8 * ... (9 - k + 2) [The first factor is 9 because a number cannot start with 0].

```
// 1. Formula solution
class Solution {
public:
    int countNumbersWithUniqueDigits(int n) {
```

```
        if (n == 0) return 1;
        int res = 0;
        for (int i = 1; i <= n; ++i) {
            res += count(i);
        }
        return res;
    }

    int count(int k) {
        if (k == 1) return 10;
        int res = 1;
        for (int i = 9; i >= (9 - k + 2); --i) {
            res *= i;
        }
        return res * 9;
    }
};


// 2. Backtracking solution
class Solution {
public:
    int countNumbersWithUniqueDigits(int n) {
        int res = 1, max = pow(10, n), used = 0;
        for (int i = 1; i < 10; ++i) {
            used |= (1 << i);
            res += search(i, max, used);
            used &= ~(1 << i);
        }
        return res;
    }
    int search(int pre, int max, int used) {
        int res = 0;
        if (pre < max) ++res;
        else return res;
        for (int i = 0; i < 10; ++i) {
            if (!(used & (1 << i))) {
                used |= (1 << i);
                int cur = 10 * pre + i;
                res += search(cur, max, used);
                used &= ~(1 << i);
            }
        }
        return res;
    }
};
```

1.10 Integer Break (M)

Given a positive integer n, break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given n = 2, return 1 (2 = 1 + 1); given n = 10, return 36 (10 = 3 + 3 + 4).

Note: You may assume that n is not less than 2 and not larger than 58.

Hint:
There is a simple O(n) solution to this problem.
You may check the breaking results of n ranging from 7 to 10 to discover the regularities.

```
/** 2 = 1 + 1
 *  3 = 2 + 1
 *  4 = 2 + 2
 *  5 = 3 + 2
 *  6 = 3 + 3
 *  7 = 3 + 4
 *  8 = 3 + 3 + 2
 *  9 = 3 + 3 + 3
 *  10 = 3 + 3 + 4
 */
class Solution {
public:
    int integerBreak(int n) {
        if (n == 2 || n == 3) return n - 1;
        int res = 1;
        while (n > 4) {
            res *= 3;
            n -= 3;
        }
        return res * n;
    }
};
```

1.11 Plus One (E)

Given a non-negative number represented as an array of digits, plus one to the number. The digits are stored such that the most significant digit is at the head of the list.

```
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
```

```
        int tmp = 1;

        for (int i = digits.size()-1; i >= 0; --i) {
            digits[i] += tmp;
            if (digits[i] == 10) {
                digits[i] = 0;
                tmp = 1;
            } else {
                tmp = 0;
            }
        }

        if (tmp == 1) {
            digits[0] = 1;
            digits.push_back(0);
        }

        return digits;
    }
};
```

1.12 Add Digits (E)

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example: Given num = 38, the process is like: $3 + 8 = 11$, $1 + 1 = 2$. Since 2 has only one digit, return it.

Follow up: Could you do it without any loop/recursion in O(1) runtime?

```
class Solution {
public:
    int addDigits(int num) {
        if (num < 10)
            return num;

        int sum = 0;
        while (num != 0) {
            sum += num % 10;
            num /= 10;
        }

        return addDigits(sum);
    }
```

```
};
```

1.13 Happy Number (E)

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example: 19 is a happy number

$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

```
class Solution {
public:
    bool isHappy(int n) {
        if (n == 1)
            return true;
        // All non-happy numbers follow sequences that reach the cycle: 4, 16,
            37, 58, 89, 145, 42, 20, 4, ...
        if (n == 4)
            return false;

        int digit = 0;
        while (n != 0) {
            digit += pow((n % 10), 2);
            n /= 10;
        }

        return isHappy(digit);
    }
};
```

1.14 Ugly Number (E)

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

```
class Solution {
public:
    bool isUgly(int num) {
        if (num == 0)
            return false;

        while (num % 2 == 0)
            num /= 2;

        while (num % 3 == 0)
            num /= 3;

        while (num % 5 == 0)
            num /= 5;

        return num == 1;
    }
};
```

1.15 Ugly Number II (M)

Write a program to find the n-th ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

Note that 1 is typically treated as an ugly number.

```
class Solution {
public:
    int nthUglyNumber(int n) {
        if (n <= 0)
            return 0;

        int idx2 = 0, idx3 = 0, idx5 = 0;
        vector<int> res(n);
        res[0] = 1;

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

```
            res[i] = min(2 * res[idx2], min(3 * res[idx3], 5 * res[idx5]));
            if (res[i] == 2 * res[idx2])
                ++idx2;
            if (res[i] == 3 * res[idx3])
                ++idx3;
            if (res[i] == 5 * res[idx5])
                ++idx5;
        }

        return res[n-1];
    }
};
```

1.16 Super Ugly Number (M)

Write a program to find the n-th super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

Note: (1) 1 is a super ugly number for any given primes. (2) The given numbers in primes are in ascending order. (3) $0 < k \leq 100, 0 < n \leq 10^6, 0 < primes[i] < 1000$.

```
class Solution {
public:
    int nthSuperUglyNumber(int n, vector<int>& primes) {
        vector<int> res(n, INT_MAX);
        vector<int> index(primes.size(), 0);
        res[0] = 1;

        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < primes.size(); ++j) {
                res[i] = min(res[i], primes[j] * res[index[j]]);
            }
            for (int j = 0; j < index.size(); ++j) {
                if (res[i] == primes[j] * res[index[j]]) {
                    ++index[j];
                }
            }
        }

        return res[n-1];
    }
};
```

1.17 Count Primes (E)

Count the number of prime numbers less than a non-negative number, n.
A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself.

```
class Solution {
public:
    int countPrimes(int n) {
        if (n <= 0)
            return 0;

        bool isPrime[n];
        for (int i = 2; i < n; ++i)
            isPrime[i] = true;

        // if i is a prime, then i*i, i*(i+1), i*(i+2), ... are not prime
        for (int i = 2; i * i < n; ++i) {
            if (isPrime[i] == false)
                continue;
            for (int j = i * i; j < n; j += i)
                isPrime[j] = false;
        }

        int cnt = 0;
        for (int i = 2; i < n; ++i){
            if (isPrime[i] == true)
                ++cnt;
        }
        return cnt;
    }
};
```

1.18 Pow(x, n) (M)

Implement pow(x, n).

```
// x^n = x^(n/2) * x^(n/2) if n is even
// x^n = x * x^(n/2) * x^(n/2) if n is odd
class Solution {
public:
```

```
    double myPow(double x, int n) {
        if (n < 0) return 1 / power(x, -n);
        else return power(x, n);
    }
    double power(double x, int n) {
        if (n == 0) return 1;
        double half = power(x, n/2);
        if (n % 2 == 0) return half * half;
        else return x * half * half;
    }
};
```

1.19 Super Pow (M)

Your task is to calculate ab mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

Example1: a = 2 b = [3] Result: 8

Example2: a = 2 b = [1,0] Result: 1024

```
class Solution {
public:
    int superPow(int a, vector<int>& b) {
        long long res = 1;
        for (int i = 0; i < b.size(); ++i) {
            //e.g. 2^23 % 1337 = (2^2)^10 * 2^3 % 1337
            res = power(res, 10) * power(a, b[i]) % 1337;
        }
        return res;
    }
    int power(int x, int n) {
        if (n == 0) return 1;
        if (n == 1) return x % 1337;
        return power(x % 1337, n / 2) * power(x % 1337, n - n / 2) % 1337;
    }
};
```

1.20 Sqrt(x) (M)

Implement int sqrt(int x). Compute and return the square root of x.

```
// Find a candidate sq and decide the search range based on sq and x
class Solution {
public:
    int mySqrt(int x) {
        long long left = 1, right = x;
        while (left <= right) {
            long long mid = left + (right - left) / 2;
            long long sq = mid * mid;
            if (sq == x) return mid;
            else if (sq < x) left = mid + 1;
            else right = mid - 1;
        }
        return right; // if x=0 or x=1, just return right
    }
};
```

1.21 Valid Perfect Square (M)

Given a positive integer num, write a function which returns True if num is a perfect square else False. Note: Do not use any built-in library function such as sqrt.

Example 1: Input: 16 Returns: True

Example 2: Input: 14 Returns: False

```
class Solution {
public:
    bool isPerfectSquare(int num) {
        long long left = 1, right = num;
        while (left <= right) {
            long long mid = left + (right - left) / 2;
            long long sq = mid * mid;
            if (sq == num) return true;
            else if (sq < num) left = mid + 1;
            else right = mid - 1;
        }
        return false;
    }
};
```

Chapter 2

Bit Manipulation

2.1 Sum of Two Integers (E)

Calculate the sum of two integers a and b, but you are not allowed to use the operator + and -.

Example:
Given a = 1 and b = 2, return 3.

```
class Solution {
public:
    int getSum(int a, int b) {
        int sum = a;
        while (b != 0) {
            sum = a ^ b;              // use ^ to find different bits
            b = (a & b) << 1;         // use & to find carry, then shift one
                position left
            a = sum;
        }
        return sum;
    }
};
```

2.2 Grey Code (M)

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:
00 - 0

01 - 1
11 - 3
10 - 2

```
// Binary to grey code
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res;
        for (int i = 0; i < pow(2,n); ++i) {
            res.push_back((i >> 1) ^ i);
        }
        return res;
    }
};
```

2.3 Power of Two (E)

Given an integer, write a function to determine if it is a power of two.

```
class Solution {
public:
    bool isPowerOfTwo(int n) {   // 2^x = n
        return (n > 0) && (n & (n-1)) == 0;
    }
};
```

2.4 Power of Three (E)

Given an integer, write a function to determine if it is a power of three.

```
class Solution {
public:
    bool isPowerOfThree(int n) {
        if (n <= 0)
            return false;

        while(n % 3 == 0)
            n /= 3;

        return n == 1;
    }
```

```
};
```

2.5 Power of Four (E)

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

```
class Solution {
public:
    bool isPowerOfFour(int num) {
        return (num > 0) && ((num & (num - 1)) == 0) && ((num & 0x55555555) ==
            num);
    }
};
```

2.6 Number of 1 Bits (E)

Write a function that takes an unsigned integer and returns the number of "1" bits it has (also known as the Hamming weight).

For example, the 32-bit integer "11" has binary representation 00000000000000000000000000001011, so the function should return 3.

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int res = 0;
        while (n != 0) {
            if (n & 1 == 1)
                ++res;
            n = n >> 1;
        }
        return res;
    }
};
```

2.7 Counting Bits (M)

Given a non negative integer number num. For every numbers i in the range $0 \leq i \leq num$ calculate the number of 1's in their binary representation and return them as an array.

Example:
For num = 5 you should return [0,1,1,2,1,2].

```
class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> res = {0};
        int offset = 2;

        for (int i = 1; i <= num; ++i) {
            if (i >= offset)
                offset *= 2;
            res.push_back(res[i - offset/2] + 1);   // dp[index] = dp[index -
                offset] + 1;
        }

        return res;
    }
};
```

2.8 Reverse Bits (E)

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as 00000010100101000001111010011100), return 964176192 (represented in binary as 00111001011110000010100101000000).

Follow up:
If this function is called many times, how would you optimize it?

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        int res = 0;
        for (int i = 0; i < 32; ++i){
            res += n & 1;     // assign the right most bit of n to res
            n >>= 1;          // remove the assigned bit from n
            if (i < 31)
                res <<= 1;    // move the assigned bit in res to left for
                    reversing
        }
        return res;
    }
};
```

2.9 Single Number (E)

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

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

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int result = 0;
        for (int i = 0; i < nums.size(); ++i)
            result ^= nums[i];
        return result;
    }
};
```

2.10 Single Number II (M)

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

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

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int x1 = 0;
        int x2 = 0;
        int mask = 0;

        for (int i = 0; i < nums.size(); ++i) {
            x2 ^= x1 & nums[i];
            x1 ^= nums[i];
            mask = ~(x1 & x2);
            x2 &= mask;
            x1 &= mask;
     }
```

```
    return x1;
    }
};
```

2.11 Single Number III (M)

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example: Given nums = [1, 2, 1, 3, 2, 5], return [3, 5].

Note: The order of the result is not important. So in the above example, [5, 3] is also correct. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

```
class Solution {
public:
    vector<int> singleNumber(vector<int>& nums) {
        int r = 0, n = nums.size(), i = 0, last = 0;
        vector<int> ret(2, 0);

        for (i = 0; i < n; ++i)
            r ^= nums[i];         // r = A ^ B

        last = r & (~(r - 1));    // get the last '1'

        for (i = 0; i < n; ++i)
        {
            if ((last & nums[i]) != 0)
                ret[0] ^= nums[i];       // group with the same position of '1'
            else
                ret[1] ^= nums[i];       // group without the same position of '1'
        }

        return ret;
    }
};
```

2.12 Binary Clock (Google Interview 8.15.2016)

Suppose you have a binary clock that consists of two rows of LEDs. The first row has four LEDs representing four binary digits to indicate the hour. The bottom row has six LEDs to represent six binary digits indicating the minute. Write a program to list all the times

that consist of exactly three lights being on, and the rest being off. List the times in human readable form.

```
class BinaryClock {
public:
    void clockTime() {
        vector<string> res;
        for (int h = 0; h <= 12; ++h) {
            for (int m = 0; m < 60; ++m) {
                if (countBits(h) + countBits(m) == 3) {
                    res.push_back(to_string(h) + ':' + to_string(m));
                }
            }
        }
    }
    int countBits(int value){
      int mask = 1;
      int cnt = 0;
      for(int i = 0; i < 6; i++) {
         if((value & mask) == 1) ++cnt; // find 1, increase cnt
         mask <<= 1; // shift left to find all 1's in value
      }
      return cnt;
    }
};
```

Chapter 3

Linked List

3.1 Intersection of Two Linked Lists (E)

Write a program to find the node at which the intersection of two singly linked lists begins.

Notes:
If the two linked lists have no intersection at all, return null.
The linked lists must retain their original structure after the function returns.
You may assume there are no cycles anywhere in the entire linked structure.
Your code should preferably run in O(n) time and use only O(1) memory.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */

class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (!headA || !headB) return NULL;

        ListNode *p1 = headA;
        ListNode *p2 = headB;

        while (p1 && p2 && p1 != p2) {
            p1 = p1->next;
            p2 = p2->next;

            if (p1 == p2) return p1;
```

```
            if (!p1)   p1 = headB;
            if (!p2)   p2 = headA;
        }

        return p1;
    }
};
```

3.2 Plus One Linked List (E)

Given a non-negative number represented as a singly linked list of digits, plus one to the number. The digits are stored such that the most significant digit is at the head of the list.

Example:
Input: $1- > 2- > 3$
Output: $1- > 2- > 4$

```
class Solution {
public:
    ListNode *plusOne(ListNode *head) {
        if (!head) return head;
        int carry = 1;
        ListNode *rev_head = reverse(head), *p = rev_head;

        while (p) {
            int sum = p->val + carry;
            if (sum == 10) {
                p->val = 0;
                carry = 1;
                p = p->next;
            } else {
                carry = 0;
                break;
            }
        }

        if (carry == 1) {
            p->next = new ListNode(1);
        }

        return revese(rev_head);
    }

    ListNode *reverse(ListNode *head) {
```

```
        ListNode *p1 = head->next, *p2;
        head->next = NULL;

        while (p1->next) {
            p2 = p1->next;
            p1->next = head;
            head = p1;
            p1 = p2;
        }

        return head;
    }
};
```

3.3 Reverse Linked List (E)

Reverse a singly linked list.

```
// iterative solution
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == NULL || head->next == NULL)
            return head;

        ListNode *p1 = head->next, *p2;
        head->next = NULL;
        while (p1 != NULL) {
            p2 = p1->next;
            p1->next = head;
            head = p1;
            p1 = p2;
        }

        return head;
    }
};

// recursive solution
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (!head || !(head -> next))
            return head;
        ListNode* node = reverseList(head -> next);
```

```
        head -> next -> next = head;
        head -> next = NULL;
        return node;
    }
};
```

3.4 Rotate Linked List (E)

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:
Given $1- > 2- > 3- > 4- > 5- > NULL$ and k = 2, return $4- > 5- > 1- > 2- > 3- > NULL$.

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || k <= 0) return head;

        ListNode *p = head;
        int n = 1;

        while (p->next){      // get the length of list
            p = p->next;
            ++n;
        }

        p->next = head;       // make a circle list
        k = k % n;            // get the correct k

        for(int i = 0; i < n - k; ++i) // move p to the correct position
            p = p->next;

        head = p->next;
        p->next = NULL;
        return head;
    }
};
```

3.5 Palindrome Linked List (E)

Given a singly linked list, determine if it is a palindrome.

```
// reverse the right half list and compare it to the left half list
class Solution {
public:
    bool isPalindrome(ListNode *head) {
        if (!head || !head->next) return true;

        ListNode *slow = head, *fast = head;
        while (fast->next && fast->next->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        slow->next = reverseList(slow->next);
        slow = slow->next;

        while (slow) {
            if (head->val != slow->val) return false;
            head = head->next;
            slow = slow->next;
        }

        return true;
    }

    ListNode *reverseList(ListNode *head) {
        ListNode *p1 = head->next, *p2;
        head->next = NULL;

        while (p1) {
            p2 = p1->next;
            p1->next = head;
            head = p1;
            p1 = p2;
        }

        return head;
    }
};
```

3.6 Swap Nodes in Pairs (E)

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

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

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

```
// iterative
class Solution {
public:
    ListNode *swapPairs(ListNode *head) {
        ListNode *new_head = new ListNode(0);
        new_head->next = head;
        ListNode *prev = new_head, *cur = head;

        while(cur && cur->next) {
            prev->next = cur->next;
            cur->next = cur->next->next;
            prev->next->next = cur;
            prev = cur;
            cur = cur->next;
        }

        return new_head->next;
    }
};

// recursive
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if (head == NULL || head->next == NULL)
            return head;

        ListNode *p = head->next;
        head->next = swapPairs(p->next);
        p->next = head;

        return p;
    }
};
```

3.7 Odd Even Linked List (M)

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example: Given $1- > 2- > 3- > 4- > 5- > NULL$, return $1- > 3- > 5- > 2- > 4- > NULL$.

Note: The relative order inside both the even and odd groups should remain as it was in the input. The first node is considered odd, the second node even and so on ...

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *odd = head, *even = head->next;
        while (even && even->next) {
            ListNode *tmp = odd->next; // must store odd->next as tmp
            odd->next = even->next;
            even->next = even->next->next;
            odd->next->next = tmp; // update odd->next->next to tmp but not the
                current even
            even = even->next;
            odd = odd->next;
        }
        return head;
    }
};

class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *odd = head, *even = head->next, *even_head = even;
        while (even && even->next) {
            odd->next = even->next;
            odd = even->next;
            even->next = odd->next;
            even = odd->next;
        }
        odd->next = even_head;
        return head;
    }

};
```

3.8 Delete Node in a Linked List (E)

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is 1-2-3-4 and you are given the third node with value 3, the linked list should become 1-2-4 after calling your function.

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        ListNode *tmp = node->next;
        *node = *tmp;
        delete tmp;
    }
};
```

3.9 Remove Linked List Elements (E)

Remove all elements from a linked list of integers that have value val.

Example:
Given: 1-2-6-3-4-5-6, val = 6
Return: 1-2-3-4-5

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        while (head != NULL && head->val == val)
            head = head->next;

        if (head == NULL)
            return head;

        ListNode *p1 = head, *p2 = head->next;
        while (p2 != NULL) {
            if (p2->val == val) {
                p1->next = p2->next;
            } else {
                p1 = p2;
            }
            p2 = p2->next;
        }
```

```
        return head;
    }
};
```

3.10 Remove Nth Node From End of List (E)

Given a linked list, remove the nth node from the end of list and return its head.

For example,
Given linked list: $1->2->3->4->5$, and $n = 2$.
After removing the second node from the end, the linked list becomes $1->2->3->5$.

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* slow = head;
        ListNode* fast = head;

        for (int i = 1; i <= n; ++i)
            fast = fast->next;

        if (!fast) return head->next;

        while (fast->next) {
            slow = slow->next;
            fast = fast->next;
        }
        slow->next = slow->next->next;   // remove the n-th node

        return head;
    }
};
```

3.11 Remove Duplicates from Sorted List (E)

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example:
Given 1-1-2, return 1-2.
Given 1-1-2-3-3, return 1-2-3.

```
class Solution {
```

```
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == NULL || head->next == NULL)
            return head;

        ListNode *p1 = head, *p2 = head->next;

        while (p2 != NULL) {
            if (p1->val == p2->val) {
                p1->next = p2->next;
            } else {
                p1 = p2;
            }
            p2 = p2->next;
        }

        return head;
    }
};
```

3.12 Remove Duplicates from Sorted List II (M)

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,
Given 1-2-3-3-4-4-5, return 1-2-5.
Given 1-1-1-2-3, return 2-3.

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == NULL || head->next == NULL)
            return head;

        ListNode *new_head = new ListNode(0); // define a new head
        new_head->next = head;
        ListNode *p1 = new_head, *p2 = head;

        while(p2 != NULL) {
            while (p2->next != NULL && p2->val == p2->next->val)  // set p2 to
                the last node of duplicates
                p2 = p2->next;

            if (p1->next == p2) {
```

```
            p1 = p1->next;              // no duplicate
        } else {
            p1->next = p2->next;        // skip all duplicates
        }

        p2 = p2->next;
    }

    return new_head->next;

  }
};
```

3.13 Linked List Cycle (E)

Given a linked list, determine if it has a cycle in it.

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (head == NULL)
            return false;

        ListNode *slow = head, *fast = head;
        while (fast->next != NULL && fast->next->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast)
                return true;
        }

        return false;
    }
};
```

3.14 Linked List Cycle II (M)

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

```
/**
 * len: length of the linked list
 * a: length of the head to the node where the cycle begins
```

```
 * b: length of the node where the cycle begins to the node where the slow and
    fast meet
 * r: length of the cycle
 * s: length of the slow moved
 * s = a + b, 2s = s + kr; -> a + b = kr = (k-1)r + r, r = len - a; -> a =
    (k-1)r + len - a - b
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (head == NULL)
            return NULL;

        ListNode *slow = head, *fast = head;

        while (fast->next != NULL && fast->next->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                fast = head;
                while (slow != fast) {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }

        return NULL;
    }
};
```

3.15 Merge Two Sorted Lists (E)

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (l1 == NULL) return l2;
        if (l2 == NULL) return l1;

        if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
```

```
            return l1;
        } else {
            l2->next = mergeTwoLists(l2->next, l1);
            return l2;
        }
    }
};
```

3.16 Merge k Sorted Lists (H)

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

```
// Divide & Conquer
// Suppose initially each list is of average length n, then: k/2*(2n) + k/4*(4n)
    + k/8*(8n)... + = logk * (kn)
class Solution {
public:
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        if(lists.empty()) return NULL;

        while(lists.size() > 1){
            lists.push_back(mergeTwoLists(lists[0], lists[1]));
            lists.erase(lists.begin());
            lists.erase(lists.begin());
        }
        return lists.front();
    }

    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (l1 == NULL) return l2;
        if (l2 == NULL) return l1;

        if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l2->next, l1);
            return l2;
        }
    }
};
```

3.17 Flatten Nested List Iterator (M)

Given a nested list of integers, implement an iterator to flatten it. Each element is either an integer, or a list – whose elements may also be integers or other lists.

Example 1: Given the list [[1,1],2,[1,1]],
By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].

Example 2: Given the list [1,[4,[6]]],
By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,4,6].

```
/**
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * class NestedInteger {
 *   public:
 *     // Return true if this NestedInteger holds a single integer, rather than a
    nested list.
 *     bool isInteger() const;
 *
 *     // Return the single integer that this NestedInteger holds, if it holds a
    single integer
 *     // The result is undefined if this NestedInteger holds a nested list
 *     int getInteger() const;
 *
 *     // Return the nested list that this NestedInteger holds, if it holds a
    nested list
 *     // The result is undefined if this NestedInteger holds a single integer
 *     const vector<NestedInteger> &getList() const;
 * };
 */

class NestedIterator {
public:
    NestedIterator(vector<NestedInteger> &nestedList) {
        for (int i = nestedList.size() - 1; i >= 0; --i)
            s.push(nestedList[i]);
    }

    int next() {
        NestedInteger n = s.top();
        s.pop();
        return n.getInteger();
    }
```

```
    bool hasNext() {
        while(!s.empty()) {
            NestedInteger n = s.top();
            if (n.isInteger()) return true;
            s.pop();   // pop the current element for the following push
            // push to s if the current element is a list
            for (int i = n.getList().size() - 1; i >= 0; --i)
                s.push(n.getList()[i]);
        }
        return false;
    }

private:
    stack<NestedInteger> s;
};

/**
 * Your NestedIterator object will be instantiated and called as such:
 * NestedIterator i(nestedList);
 * while (i.hasNext()) cout << i.next();
 */
```

3.18 Nested List Weight Sum (E)

Given a nested list of integers, return the sum of all integers in the list weighted by their depth. Each element is either an integer, or a list – whose elements may also be integers or other lists.

Example 1:
Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)

Example 2:
Given the list [1,[4,[6]]], return 27. (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4*2 + 6*3 = 27)

```
class Solution {
public:
    int depthSum(vector<NestedInteger>& nestedList) {
        int res = 0;
        for (auto l : nestedList)
            res += getSum(l, 1);
        return res;
    }
```

```
    int getSum(NestedInteger nl, int level) {
        int res = 0;
        if (nl.isInteger())
            return level * nl.getInteger();
        for (auto l : nl.getList())
            res += getSum(l, level+1);
        return res;
    }
};
```

3.19 Nested List Weight Sum II (M)

Given a nested list of integers, return the sum of all integers in the list weighted by their depth. Each element is either an integer, or a list – whose elements may also be integers or other lists.

Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:
Given the list [[1,1],2,[1,1]], return 8. (four 1's at depth 1, one 2 at depth 2)

Example 2:
Given the list [1,[4,[6]]], return 17. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1*3 + 4*2 + 6*1 = 17)

```
class Solution {
public:
    int depthSumInverse(vector<NestedInteger> &nestedList) {
        int unweighted = 0, weighted = 0;

        while (!nestedList.empty()) {
            vector<NestedInteger> nextLevel;
            for (auto l : nestedList) {
                if (l.isInteger()) {
                    unweighted += l.getInteger();    // add all integers of the
                        current level togeter
                } else {
                    // insert(position, first, last)
                    nextLevel.insert(nextLevel.end(), l.getList().begin(),
                        l.getList().end());
                }
```

```
            }
            weighted += unweighted;  // add all integers in level n to level n+1
                twice
            nestedList = nextLevel;  // update the nested list
        }

        return weighted;
    }
};
```

Chapter 4

Stack and Queue

4.1 Implement Stack using Queues (E)

Implement the following operations of a stack using queues.

push(x) – Push element x onto stack.
pop() – Removes the element on top of the stack.
top() – Get the top element.
empty() – Return whether the stack is empty.

```
class Stack {
queue<int> que;
public:
    // Push element x onto stack.
    void push(int x) {
       que.push(x);                          // push x to tail
       for (int i = 1; i < que.size(); ++i) { // repeat until x is the head
           que.push(que.front());            // push head to tail
           que.pop();                        // pop the old head
       }
    }

    // Removes the element on top of the stack.
    void pop() {
        que.pop();
    }

    // Get the top element.
    int top() {
        return que.front();
    }

    // Return whether the stack is empty.
```

```
    bool empty() {
        return que.empty();
    }
};
```

4.2 Implement Queue using Stacks (E)

Implement the following operations of a queue using stacks.

push(x) – Push element x to the back of queue.
pop() – Removes the element from in front of queue.
peek() – Get the front element.
empty() – Return whether the queue is empty.

```
class Queue {
stack<int> s1, s2;
public:
    // Push element x to the back of queue.
    void push(int x) {
        while (!s2.empty()){
            s1.push(s2.top());
            s2.pop();
        }
        s1.push(x);
    }

    // Removes the element from in front of queue.
    void pop(void) {
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }
        s2.pop();
    }

    // Get the front element.
    int peek(void) {
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }
        return s2.top();
    }

    // Return whether the queue is empty.
```

```
    bool empty(void) {
        return s1.empty() && s2.empty();
    }
};
```

4.3 Min Stack (E)

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

push(x) – Push element x onto stack.
pop() – Removes the element on top of the stack.
top() – Get the top element.
getMin() – Retrieve the minimum element in the stack.

Example:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); − > Returns -3.
minStack.pop();
minStack.top(); − > Returns 0.
minStack.getMin(); − > Returns -2.

```
class MinStack {
public:
    stack<int> s;
    stack<int> s_min;                    // use a new stack to store minimum
        numbers

    void push(int x) {
        s.push(x);
        if (s_min.empty() || x <= getMin()) // make a copy of a min value to s_min
            s_min.push(x);
    }

    void pop() {
        if (s.top() == getMin())         // update s_min if the min value is poped
            s_min.pop();
        s.pop();
    }

    int top() {
```

```
        return s.top();
    }

    int getMin() {
        return s_min.top();
    }
};
```

4.4 Moving Average from Data Stream (E)

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

For example,
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3

```
class MovingAverage {
public:
    MovingAverage(int size) {
        this->size = size;
        sum = 0;
    }

    double next(int val) {
        if (q.size() >= size) {
            sum -= q.front();
            q.pop();
        }
        q.push(val);
        sum += val;
        return sum / q.size();
    }

private:
    queue<int> q;
    int size;
    double sum;
};
```

Chapter 5

Array

5.1 Bulls and Cows (E)

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example:
Secret number: "1807"
Friend's guess: "7810"
Hint: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.)

Please note that both secret number and friend's guess may contain duplicate digits, for example:
Secret number: "1123"
Friend's guess: "0111"
In this case, the 1st 1 in friend's guess is a bull, the 2nd or 3rd 1 is a cow, and your function should return "1A1B".

```
class Solution {
public:
    string getHint(string secret, string guess) {
        int bull = 0, cow = 0;
        int set1[10] = {0};
        int set2[10] = {0};
        int len = secret.size();

        for (int i = 0; i < len; ++i) {
            if (secret[i] == guess[i]) {
                ++bull;
```

```
            } else {
                ++set1[secret[i] - '0'];
                ++set2[guess[i] - '0'];
            }
        }

        for (int i = 0; i < 10; ++i) {
            if (set1[i] != 0 && set2[i] != 0)
                cow += min(set1[i],set2[i]);
        }

        return to_string(bull) + 'A' + to_string(cow) + 'B';
    }
};
```

5.2 Rotate Array (E)

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].

```
class Solution {
public:
    void rotate(vector<int> &nums, int k) {
        if (k == 0) return;
        int n = nums.size();
        k = k % n;            // get the correct k if k > n

        reverse(nums, 0, n-1);
        reverse(nums, 0, k-1);
        reverse(nums, k, n-1);
    }

    void reverse(vector<int> &nums, int start, int end) {
        int tmp;
        while (start < end) {
            tmp = nums[start];
            nums[start++] = nums[end];
            nums[end--] = tmp;
        }
    }
};
```

5.3 Move Zeros (E)

Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0].

Note:
You must do this in-place without making a copy of the array.
Minimize the total number of operations.

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int index = 0;
        for (int i = 0; i < nums.size(); ++i) {
            if (nums[i] != 0)
                nums[index++] = nums[i];
        }
        for (int i = index; i < nums.size(); ++i)
            nums[i] = 0;
    }
};
```

5.4 Remove Element (E)

Given an array and a value, remove all instances of that value in place and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory. The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Example:
Given input array nums = [3,2,2,3], val = 3, Your function should return length = 2, with the first two elements of nums being 2.

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int index = 0;
        for (int i = 0; i < nums.size(); ++i) {
            if (nums[i] != val)
                nums[index++] = nums[i];
```

```
        }
        return index;
    }
};
```

5.5 Remove Duplicates from Sorted Array (E)

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory.

For example,
Given input array nums = [1,1,2], your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.empty()) return 0;

        int index = 0;
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[index] != nums[i])
                nums[++index] = nums[i];
        }
        return index + 1;
    }
};
```

5.6 Missing Number (M)

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

For example, Given nums = [0, 1, 3] return 2.

Note: Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

```
class Solution {
public:
```

```
    int missingNumber(vector<int> &nums) {
        int n = nums.size(), res = (1 + n) * n / 2;
        for (int i = 0; i < n; ++i) {
            res -= nums[i];
        }
        return res;
    }
};
```

5.7 First Missing Positive (H)

Given an unsorted integer array, find the first missing positive integer.

For example, Given [1,2,0] return 3, and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int i = 0, n = nums.size();
        while (i < n) {
            if (nums[i] <= n && nums[i] > 0 && nums[i] != nums[nums[i] - 1]) {
                swap(nums[i], nums[nums[i] - 1]);
            } else {
                ++i;
            }
        }
        for (int i = 0; i < n; ++i) {
            if (nums[i] != i + 1) {
                return i + 1;
            }
        }
        return n + 1;
    }
};
```

5.8 Intersection of Two Arrays (E)

Given two arrays, write a function to compute their intersection.

Example: Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2].

Note:
Each element in the result must be unique.
The result can be in any order.

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> s(nums1.begin(), nums1.end());
        vector<int> res;

        for (auto x : nums2) {
            if (s.find(x) != s.end()) {       // Searches x in s, returns 1 if x
                is found, otherwise return 0
                res.push_back(x);
                s.erase(x);
            }
        }

        return res;
    }
};
```

5.9 Intersection of Two Arrays II (E)

Given two arrays, write a function to compute their intersection.

Example: Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].

Note:
Each element in the result should appear as many times as it shows in both arrays.
The result can be in any order.

Follow up:
What if the given array is already sorted? How would you optimize your algorithm?
What if nums1's size is small compared to nums2's size? Which algorithm is better?
What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

```
// Sol1: Hash table
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int,int> dict;
```

```
        vector<int> res;

        for (int i = 0; i < nums1.size(); ++i)
            dict[nums1[i]]++;

        for (int i = 0; i < nums2.size(); ++i){
            if (--dict[nums2[i]] >= 0)
                res.push_back(nums2[i]);
        }

        return res;
    }
};

// Sol2: Sorting & Two pointers
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        sort(nums1.begin(), nums1.end());
        sort(nums2.begin(), nums2.end());

        int n1 = nums1.size(), n2 = nums2.size();
        int i1 = 0, i2 = 0;
        vector<int> res;

        while(i1 < n1 && i2 < n2) {
            if (nums1[i1] == nums2[i2]) {
                res.push_back(nums1[i1]);
                ++i1;
                ++i2;
            } else if (nums1[i1] < nums2[i2]) {
                ++i1;
            } else {
                ++i2;
            }
        }

        return res;
    }
};
```

5.10 Merge Sorted Array (E)

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note: You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively.

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int i = m - 1;
        int j = n - 1;
        int k = m + n - 1;

        while (i >= 0 && j >= 0) {          // compare from end to start of two
            arrays
            if (nums1[i] < nums2[j]) {      // put the largest element to the end
                of array
                nums1[k--] = nums2[j--];
            } else {
                nums1[k--] = nums1[i--];
            }
        }

        while (i >= 0)                      // keep writing for the left over
            nums1[k--] = nums1[i--];

        while (j >= 0)
            nums1[k--] = nums2[j--];
    }
};
```

5.11 Sort Transformed Array (M)

Given a sorted array of integers nums and integer values a, b and c. Apply a function of the form $f(x) = ax^2 + bx + c$ to each element x in the array. The returned array must be in sorted order.

Expected time complexity: O(n)

Example:
nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5, Result: [3, 9, 15, 33]
nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5 Result: [-23, -5, 1, 7]

```
class Solution {
public:
    vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {
```

```
        int n = nums.size(), i = 0, j = n - 1, k;
        vector<int> res(n);
        // Decide which side we should write sorted result to res
        if (a >= 0) {  // parabola open up
            k = n - 1;
        } else {       // parabola open down
            k = 0;
        }
        while (i <= j) {
            if (a >= 0) { // large number first
                if (cal(nums[i],a,b,c) >= cal(nums[j],a,b,c)) {
                    res[j--] = cal(nums[i++],a,b,c);
                } else {
                    res[j--] = cal(nums[j--],a,b,c);
                }
            } else { // small number first
                if (cal(nums[i],a,b,c) >= cal(nums[j],a,b,c)) {
                    res[i++] = cal(nums[j--],a,b,c);
                } else {
                    res[i++] = cal(nums[i++],a,b,c);
                }
            }
        }
        return res;
    }
    int cal(int x, int a, int b, int c) {
        return a * x * x + b * x + c;
    }
};
```

5.12 Majority Element (E)

Given an array of size n, find the majority element. The majority element is the element that appears more than floor(n/2) times.

You may assume that the array is non-empty and the majority element always exist in the array.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int element, counts = 0;

        for (int i = 0; i < nums.size(); ++i) {
            if (counts == 0) {
```

```
                element = nums[i];
                ++counts;
            } else {
                if (element == nums[i])
                    ++counts;
                else
                    --counts;
            }
        }

        return element;
    }
};
```

5.13 Majority Element II (M)

Given an integer array of size n, find all elements that appear more than floor(n/3) times. The algorithm should run in linear time and in O(1) space.

```
class Solution {
public:
    vector<int> majorityElement(vector<int>& nums) {
        int candidate1 = 0, candidate2 = 0, count1 = 0, count2 = 0;
        vector<int> res;

        // 1. get candidates
        for (auto n : nums) {
            if (candidate1 == n) {
                ++count1;
            } else if (candidate2 == n) {
                ++count2;
            } else if (count1 == 0) {
                candidate1 = n;
                ++count1;
            } else if (count2 == 0) {
                candidate2 = n;
                ++count2;
            } else {
                --count1;
                --count2;
            }
        }

        // 2. get the count of each candidate
        count1 = 0;
```

```
        count2 = 0;
        for (auto n : nums) {
            if (candidate1 == n)
                ++count1;
            else if (candidate2 == n)
                ++count2;
        }

        // 3. check if each candidate satisfies the majority condition
        if (count1 > nums.size() / 3)
            res.push_back(candidate1);
        if (count2 > nums.size() / 3)
            res.push_back(candidate2);

        return res;
    }
};
```

5.14 Contains Duplicate (E)

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

```
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        unordered_set<int> s;
        for (int i = 0; i < nums.size(); ++i) {
            if (s.find(nums[i]) != s.end())
                return true;
            s.insert(nums[i]);
        }
        return false;
    }
};
```

5.15 Contains Duplicate II (E)

Given an array of integers and an integer k, find out whether there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k.

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        unordered_set<int> s;

        if (k <= 0)
            return false;
        if (k >= nums.size())
            k = nums.size() - 1;

        for (int i = 0; i < nums.size(); ++i) {
            if (i > k)
                s.erase(nums[i-k-1]);
            if (s.find(nums[i]) != s.end())
                return true;
            s.insert(nums[i]);
        }

        return false;
    }
};
```

5.16 Contains Duplicate III (M)

Given an array of integers, find out whether there are two distinct indices i and j in the array such that the difference between nums[i] and nums[j] is at most t and the difference between i and j is at most k.

```
class Solution {
public:
    bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
        set<int> s; // set is ordered automatically

        for (int i = 0; i < nums.size(); i++) {
            // keep the set contains nums i j at most k
            if (i > k)
                s(nums[i-k-1]);

            // |x - nums[i]| <= t ==> -t <= x - nums[i] <= t;
            auto pos = s(nums[i] - t); // x-nums[i] >= -t ==> x >= nums[i]-t
            if (pos != s() && *pos - nums[i] <= t) // x - nums[i] <= t
                return true;

            s(nums[i]);
        }
```

```
        return false;
    }
};
```

5.17 Find the Duplicate Number (H)

Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note: You must not modify the array (assume the array is read only).
You must use only constant, O(1) extra space.
Your runtime complexity should be less than $O(n^2)$.
There is only one duplicate number in the array, but it could be repeated more than once.

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int low = 1, high = nums.size() - 1;
        // Use the mid of 1~n, not the mid of nums[0]~nums[nums.size()-1]
        while (low < high) {
            int mid = low + (high - low) / 2;
            int cnt = 0;
            for (int i = 0; i < nums.size(); ++i) {
                if (nums[i] <= mid) {
                    ++cnt;
                }
            }
            if (cnt <= mid) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
};
```

5.18 Top K Frequent Elements (M)

Given a non-empty array of integers, return the k most frequent elements.

For example, Given [1,1,1,2,2,3] and k = 2, return [1,2].

Note: You may assume k is always valid, $1 \leq k \leq$ number of unique elements.
Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> m;
        // Priority queues are a type of container adaptors, specifically
            designed such that its first element is always the greatest of the
            elements it contains, according to some strict weak ordering criterion.
        priority_queue<pair<int, int>> q;
        vector<int> res;
        for (auto a : nums) {
            ++m[a];
        }
        for (auto it : m) {
            q.push({it.second, it.first});
        }
        for (int i = 0; i < k; ++i) {
            res.push_back(q.top().second);
            q.pop();
        }
        return res;
    }
};
```

5.19 Two Sum (E)

Given an array of integers, return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution.

Example:
Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

```
class Solution {
public:
    vector<int> twoSum(vector<int> &nums, int target) {
        vector<int> res;
        unordered_map<int, int> map;

        for (int i = 0; i < nums.size(); ++i)
```

```
            map[nums[i]] = i;

        for (int i = 0; i < nums.size(); ++i) {
            int tmp = target - nums[i];
            if (map.find(tmp) != map.end() && i != map[tmp]){
                res.push_back(i);
                res.push_back(map[tmp]);
                break;
            }
        }

        return res;
    }
};
```

5.20 Two Sum II (M)

Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: numbers=2, 7, 11, 15, target=9
Output: index1=1, index2=2

```
class Solution {
public:
    vector<int> twoSum(vector<int> &nums, int target) {
        vector<int> res;
        int left = 0, right = nums.size() - 1;

        while (left < right) {
            int tmp = nums[left] + nums[right];
            if (tmp < target) {
                ++left;
            } else if (tmp > target) {
                --right;
            } else {
                res.push_bcak(left+1);
                res.push_back(right+1);
```

```
                return res;
            }
        }
    }
};
```

5.21 Two Sum III (E)

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure.
find - Find if there exists any pair of numbers which sum is equal to the value.

For example,
add(1); add(3); add(5);
find(4) $->$ true
find(7) $->$ false

```
class Solution {
public:
    unordered_map<int, int> map;

    void add(int num) {
        ++map[num];
    }

    bool find(int num) {
        for (auto i : map) {
            int tmp = num - i.first;
            if (map.find(tmp) != map.end()) {
                if (tmp != i.first)          // the pair of numbers has two
                    different numbers
                    return true;
                else if (i.second >= 2)      // the pair of numbers has two same
                    numbers
                    return true;
            }
        }
        return false;
    }
};
```

5.22 3Sum (M)

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. 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]]

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> res;
        if (nums.size() < 3) return res;
        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size()-2; ++i) {
            if (i != 0 && nums[i] == nums[i-1]) continue;      // skip duplicates

            // like the two sum problem, let j to be left, and k to be right
            int j = i + 1, k = nums.size() - 1;
            while (j < k) {
                if (nums[i] + nums[j] + nums[k] == 0) {
                    res.push_back({nums[i], nums[j], nums[k]});
                    ++j;
                    --k;
                    while (j < k && nums[j] == nums[j-1]) ++j; // skip duplicates
                    while (j < k && nums[k] == nums[k+1]) --k; // skip duplicates
                } else if (nums[i] + nums[j] + nums[k] < 0) {
                    ++j;
                    while (j < k && nums[j] == nums[j-1]) ++j; // skip duplicates
                } else {
                    --k;
                    while (j < k && nums[k] == nums[k+1]) --k; // skip duplicates
                }
            }
        }

        return res;
    }
};
```

5.23 3Sum Closest (M)

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = -1 2 1 -4, and target = 1. The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

```
class Solution {
public:
    int threeSumClosest(vector<int> &nums, int target) {
        if (nums.size() < 3)  return 0;
        int res = 0, min = INT_MAX;
        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size() - 2; ++i) {
            int j = i + 1, k = nums.size() - 1;
            while (j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                int diff = abs(sum - target);
                if (diff < min) {        // find the closest sum
                    min = diff;
                    res = sum;
                }
                if (sum < target)        // update j and k
                    ++j;
                else if (sum > target)
                    --k;
                else
                    return res;
            }
        }
        return res;
    }
};
```

5.24 3Sum Smaller (M)

Given an array of n integers nums and a target, find the number of index triplets i, j, k with $0 <= i < j < k < n$ that satisfy the condition $nums[i] + nums[j] + nums[k] < target$.

For example, given nums = [-2, 0, 1, 3], and target = 2.

Return 2. Because there are two triplets which sums are less than 2: [-2, 0, 1] [-2, 0, 3]

```
class Solution {
public:
    int threeSumSmaller(vector<int> &nums, int target) {
        if (nums.size() < 3)  return 0;
        int cnt = 0;
        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size() - 2; ++i){
            int j = i + 1; k = nums.size() - 1;
            while (j < k) {
                int sum = nums[i] + nums[j] + nums[k]
                if (sum >= target) {
                    --k;
                } else {
                    cnt += k - j; // if j is fixed, all numbers from j to k (less
                        than k, not include j) are satisfied
                    ++j;          // update j for the next iteration
                }
            }
        }

        return cnt;
    }
};
```

5.25 4Sum (M)

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target. The solution set must not contain duplicate quadruplets.

For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.
A solution set is: [[-1, 0, 0, 1], [-2, -1, 1, 2], [-2, 0, 0, 2]]

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        vector<vector<int>> res;
        if (nums.size() < 4)  return res;
        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size() - 3; ++i) {
```

```
            if (i != 0 && nums[i] == nums[i-1]) continue;
            for (int j = i + 1; j < nums.size() - 2; ++j) {
                if (j != i + 1 && nums[j] == nums[j-1]) continue;
                int p = j + 1, q = nums.size() - 1;
                while (p < q) {
                    int sum = nums[i] + nums[j] + nums[p] + nums[q];
                    if (sum < target) {
                        ++p;
                        while (p < q && nums[p] == nums[p-1]) ++p;
                    } else if (sum > target) {
                        --q;
                        while (p < q && nums[q] == nums[q+1]) --q;
                    } else {
                        res.push_back({nums[i], nums[j], nums[p], nums[q]});
                        ++p;
                        --q;
                        while (p < q && nums[p] == nums[p-1]) ++p;
                        while (p < q && nums[q] == nums[q+1]) --q;
                    }
                }
            }
        }
        return res;
    }
};
```

5.26 Maximum Subarray (M)

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.

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if (nums.empty()) return 0;
        int res = INT_MIN, tmp = 0;
        for (int i = 0; i < nums.size(); ++i) {
            tmp = max(nums[i], tmp + nums[i]);
            res = max(res, tmp);
        }
        return res;
    }
```

```
};
```

5.27 Range Addition (M)

Assume you have an array of length n initialized with all 0's and are given k update operations. Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc. Return the modified array after all k operations were executed.

Example: Given: length = 5, updates = [[1, 3, 2], [2, 4, 3], [0, 2, -2]]

Output: [-2, 0, 3, 5, 3]

Explanation:

Initial state: [0, 0, 0, 0, 0]

After applying operation [1, 3, 2]: [0, 2, 2, 2, 0]

After applying operation [2, 4, 3]: [0, 2, 5, 5, 3]

After applying operation [0, 2, -2]: [-2, 0, 3, 5, 3]

Hint:
Thinking of using advanced data structures? You are thinking it too complicated.
For each update operation, do you really need to update all elements between i and j?
Update only the first and end element is sufficient.
The optimal time complexity is O(k + n) and uses O(1) extra space.

```
class Solution {
public:
    vector<int> getModifiedArray(int length, vector<vector<int>>& updates) {
        vector<int> res(length+1, 0);

        // 1. The update query (l, r, v) requires that arr[i] += v for i in l..r
            (both l and r inclusive).
        // 2. Applying the final transformation [3] ensures that the increment of
            +v on arr[l] is carried through to all arr[i] for i >= l.
        // 3. The increment of -v on arr[r+1] ensures that the previous +v
            increment is cancelled out for each arr[i] for i >= r+1.
        for (int i = 0; i < update.size(); ++i) {
            res[update[i][0]] += update[i][2];
            res[update[i][1] + 1] -= update[i][2];
```

```
        }

        // propagate the addition number to all corresponding positions except
            the last one
        for (int i = 1; i < length; ++i) {
            res[i] += res[i-1];
        }

        res.pop_back(); // remove the last element
        return res;
    }
};
```

5.28 Maximum Product Subarray (M)

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.

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int mx = 1, mn = 1, res = INT_MIN;
        for (int i = 0; i < nums.size(); ++i) {
            int tmp = mx; // keep the previous max to avoid overwriting of max
                value
            mx = max(max(nums[i], mx * nums[i]), mn * nums[i]);
            mn = min(min(nums[i], tmp * nums[i]), mn * nums[i]); // use the prev
                max here
            res = max(mx, res);
        }
        return res;
    }
};
```

5.29 Product of Array Except Self (M)

Given an array of n integers where n > 1, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i]. Solve it without division and in O(n).

For example, given [1,2,3,4], return [24,12,8,6].

Follow up: Could you solve it with constant space complexity? (Note: The output array does not count as extra space for the purpose of space complexity analysis.)

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int> &nums) {
        int n = nums.size(), tmp = 1;
        vector<int> res(n, 1);

        // production of all elements before i
        for (int i = 1; i < n; ++i) {
            res[i] = res[i-1] * nums[i-1];
        }

        // production of all elements after i
        for (int i = n-1; i >= 0; --i) {
            res[i] *= tmp;    // the tmp element after the last i is set to 1
            tmp *= nums[i];   // update tmp during each iteration
        }

        return res;
    }
};
```

5.30 Maximum Size Subarray Sum Equals k (M)

Given an array nums and a target value k, find the maximum length of a subarray that sums to k. If there isn't one, return 0 instead.

Example 1:
Given nums = [1, -1, 5, -2, 3], k = 3, return 4. (because the subarray [1, -1, 5, -2] sums to 3 and is the longest)

Example 2:
Given nums = [-2, -1, 2, 1], k = 1, return 2. (because the subarray [-1, 2] sums to 1 and is the longest)

Follow Up: Can you do it in O(n) time?

```
class Solution {
public:
```

```
    int maxSubArrayLen(vector<int>& nums, int k) {
        int sum = 0, res = 0;
        unordered_map<int, int> m;
        for (int i = 0; i < nums.size(); ++i) {
            sum += nums[i];
            if (sum == k) res = i + 1;
            // if sum - prev_sum = k, i.e. sum - k = prev_sum,
            // we can remove prev_sum from sum to get the subarray sum equals k
            else if (m.count(sum-k)) res = max(res, i - m[sum-k]);
            if (!m.count(sum)) m[sum] = i;
        }
        return res;
    }
};
```

5.31 Minimum Size Subarray Sum (M)

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum $\geq$ 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.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int left = 0, right = 0, sum = 0, res = INT_MAX, n = nums.size();
        while (right < n) {
            // must have "right < n" to jump out the inner while loop
            while (sum < s && right < n) {
                sum += nums[right++];
            }
            while (sum >= s) {
                res = min(res, right - left);
                sum -= nums[left++];
            }
        }
        if (res == INT_MAX) return 0;
        else return res;
    }
};
```

5.32 Meeting Rooms (E)

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), determine if a person could attend all meetings.

For example, given [[0, 30],[5, 10],[15, 20]], return false.

```
/**
 * 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:
    bool canAttendMeetings(vector<Interval>& intervals) {
        sort(intervals.begin(), intervals.end(), compare);
        for (int i = 1; i < intervals.size(); ++i) {
            // the start time of i should be larger than the end time of i-1 for
                a vaild case
            if (intervals[i].start < intervals[i - 1].end) {
                return false;
            }
        }
        return true;
    }

    // use the start time to compare
    bool compare(Interval& i1, Interval& i2) {
        return i1.start < i2.start;
    }
};
```

5.33 Meeting Rooms II (M)

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.

For example, given [[0, 30],[5, 10],[15, 20]], return 2.

```
// 1. Map
```

```
class Solution {
public:
    int minMeetingRooms(vector<Interval>& intervals) {
        map<int, int> m;  // map keeps their elements ordered rather than
            unordered_map
        for (auto a : intervals) {
            ++m[a.start];
            --m[a.end];
        }
        int rooms = 0, res = 0;
        for (auto it : m) {
            rooms += it.second;
            res = max(res, rooms); // always store the max room number
        }
        return res;
    }
};

// 2. Two vectors
class Solution {
public:
    int minMeetingRooms(vector<Interval>& intervals) {
        vector<int> starts, ends;
        int res = 0, endpos = 0;
        for (auto a : intervals) {
            starts.push_back(a.start);
            ends.push_back(a.end);
        }
        sort(starts.begin(), starts.end());
        sort(ends.begin(), ends.end());
        for (int i = 0; i < intervals.size(); ++i) {
            if (starts[i] < ends[endpos]) ++res; // need a new room
            else ++endpos;
        }
        return res;
    }
};
```

5.34 Logger Rate Limiter (E)

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is not printed in the last 10 seconds. Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false. It is possible that several messages arrive roughly at the same time.

Example:
Logger logger = new Logger();

// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;

// logging string "bar" at timestamp 2
logger.shouldPrintMessage(2,"bar"); returns true;

// logging string "foo" at timestamp 3
logger.shouldPrintMessage(3,"foo"); returns false;

// logging string "bar" at timestamp 8
logger.shouldPrintMessage(8,"bar"); returns false;

// logging string "foo" at timestamp 10
logger.shouldPrintMessage(10,"foo"); returns false;

// logging string "foo" at timestamp 11
logger.shouldPrintMessage(11,"foo"); returns true;

```
class Logger {
public:
    Logger() {}

    bool shouldPrintMessage(int timestamp, string message) {
        if (!m.count(message)) {         // if the current message is not in m
            m[message] = timestamp;      // store it and return true
            return true;
        }
        if (timestamp - m[message] >= 10) { // if the current message should be
            printed
            m[message] = timestamp;      // update the timestamp of the current
                message
            return true;
        }
        return false;
    }

private:
    unordered_map<string, int> m;    // build a hashtable to store the
        message/timestamp pair
};
```

5.35 Design Hit Counter (M)

Design a hit counter which counts the number of hits received in the past 5 minutes. Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1. It is possible that several hits arrive roughly at the same time.

Example:
HitCounter counter = new HitCounter();

// hit at timestamp 1.
counter.hit(1);

// hit at timestamp 2.
counter.hit(2);

// hit at timestamp 3.
counter.hit(3);

// get hits at timestamp 4, should return 3.
counter.getHits(4);

// hit at timestamp 300.
counter.hit(300);

// get hits at timestamp 300, should return 4.
counter.getHits(300);

// get hits at timestamp 301, should return 3.
counter.getHits(301);

Follow up: What if the number of hits per second could be very large? Does your design scale?

```
// 1. Use queue
class HitCounter {
public:
    /** Initialize your data structure here. */
    HitCounter() {}

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds granularity). */
    void hit(int timestamp) {
        q.push(timestamp);
```

```
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity). */
    int getHits(int timestamp) {
        while (!q.empty() && timestamp - q.front() >= 300) {
            q.pop();
        }
        return q.size();
    }

private:
    queue<int> q;
};


// 2. Use two vectors
class HitCounter {
public:
    /** Initialize your data structure here. */
    HitCounter() {
        times.resize(300);
        hits.resize(300);
    }

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds granularity). */
    void hit(int timestamp) {
        int idx = timestamp % 300;
        if (times[idx] != timestamp) { // time limit exceeded
            times[idx] = timestamp;  // reset timestamp
            hits[idx] = 1;           // reset hit count
        } else {                     // same time stamp
            ++hits[idx];             // increase hit count
        }
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity). */
    int getHits(int timestamp) {
        int res = 0;
        for (int i = 0; i < 300; ++i) {
            if (timestamp - times[i] < 300) {
                res += hits[i];
            }
        }
        return res;
```

```
    }

private:
    vector<int> times, hits;
};
```

5.36 Flatten 2D Vector (M)

Implement an iterator to flatten a 2d vector.

For example,
Given 2d vector = [[1,2], [3], [4,5,6]]
By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,2,3,4,5,6].

Hint:
How many variables do you need to keep track?
Two variables is all you need. Try with x and y.
Beware of empty rows. It could be the first few rows.
To write correct code, think about the invariant to maintain. What is it?
The invariant is x and y must always point to a valid point in the 2d vector. Should you maintain your invariant ahead of time or right when you need it?
Not sure? Think about how you would implement hasNext(). Which is more complex?
Common logic in two different places should be refactored into a common method.

Follow up: As an added challenge, try to code it using only iterators in C++ or iterators in Java.

```
// 1. Use 1D vector
class Vector2D {
public:
    Vector2D(vector<vector<int>> &vec2d) {
        for (auto a : vec2d){
            v.insert(v.end(), a.begin(), a.end());
        }
    }
    int next() {
        return v[i++];
    }
    bool hasNext() {
        return i < v.size();
    }
private:
    vector<int> v;
```

```
    int i = 0;
};

// 2. Use two variables x and y
class Vector2D {
public:
    Vector2D(vector<vector<int>>& vec2d) {
        v = vec2d;
        x = y = 0;
    }
    int next() {
        return v[x][y++];
    }
    bool hasNext() {
        while (x < v.size() && y == v[x].size()) {
            ++x;
            y = 0;
        }
        return x < v.size();
    }
private:
    vector<vector<int>> v;
    int x, y;
};

// 3. Use iterator
class Vector2D {
public:
    Vector2D(vector<vector<int>>& vec2d) {
        x = vec2d.begin();
        end = vec2d.end();
    }
    int next() {
        return (*x)[y++];
    }
    bool hasNext() {
        while (x != end && y == (*x).size()) {
            ++x;
            y = 0;
        }
        return x != end;
    }
private:
    vector<vector<int>>::iterator x, end;
    int y = 0;
};
```

5.37 Zigzag Iterator (M)

Given two 1d vectors, implement an iterator to return their elements alternately.

For example, given two 1d vectors:
v1 = [1, 2]
v2 = [3, 4, 5, 6]
By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1, 3, 2, 4, 5, 6].

Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?

Clarification for the follow up question - Update (2015-09-18):
The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic". For example, given the following input: [[1,2,3], [4,5,6,7], [8,9]]
It should return [1,4,8,2,5,9,3,6,7].

```
// 1. Use 1D vector
class ZigzagIterator {
public:
    ZigzagIterator(vector<int> &v1, vector<int> &v2) {
        int n1 = v1.size(), n2 = v2.size(), n = max(n1, n2);
        for (int i = 0; i < n; ++i) {
            if (i < n1) v.push_back(v1[i]);
            if (i < n2) v.push_back(v2[i]);
        }
    }
    int next() {
        return v[i++];
    }
    bool hasNext() {
        return i < v.size();
    }
private:
    vector<int> v;
    int i = 0;
};

// 2. Use two variables i and j
class ZigzagIterator {
public:
    ZigzagIterator(vector<int>& v1, vector<int>& v2) {
        v.push_back(v1);
        v.push_back(v2);
```

```
        i = j = 0;
    }
    int next() {
        return i <= j ? v[0][i++] : v[1][j++];
    }
    bool hasNext() {
        if (i >= v[0].size()) i = INT_MAX;
        if (j >= v[1].size()) j = INT_MAX;
        return i < v[0].size() || j < v[1].size();
    }
private:
    vector<vector<int>> v;
    int i, j;
};

// 3. Use queue and iterator
class ZigzagIterator {
public:
    ZigzagIterator(vector<int>& v1, vector<int>& v2) {
        if (!v1.empty()) q.push(make_pair(v1.begin(), v1.end()));
        if (!v2.empty()) q.push(make_pair(v2.begin(), v2.end()));
    }
    int next() {
        auto it = q.front().first, end = q.front().second;
        q.pop();
        if (it + 1 != end) q.push(make_pair(it + 1, end));
        return *it;
    }
    bool hasNext() {
        return !q.empty();
    }
private:
    queue<pair<vector<int>::iterator, vector<int>::iterator>> q;
};
```

5.38 Sparse Matrix Multiplication (M)

Given two sparse matrices A and B, return the result of AB. You may assume that A's column number is equal to B's row number.

```
class Solution {
public:
    vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B) {
        int m = A.size(), n = B.size(), p = B[0].size();
        vector<vector<int>> res(m, vector<int>(p));
```

```
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (A[i][j] != 0) {
                    for (int k = 0; k < p; ++k) {
                        if (B[j][k] != 0) {
                            res[i][k] += A[i][j] * B[j][k];
                        }
                    }
                }
            }
        }

        return res;
    }
};
```

5.39 Set Matrix Zeroes (M)

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?

```
class Solution {
public:
    void setZeroes(vector<vector<int>> &matrix) {
        if (matrix.empty() || matrix[0].empty()) return;
        int m = matrix.size(), n = matrix[0].size();
        bool rowFlag = false, colFlag = false;

        // if the fist column has a zero element, update flag
        for (int i = 0; i < m; ++i) {
            if (matrix[i][0] == 0) colFlag = true;
        }
        // if the fist row has a zero element, update flag
        for (int j = 0; j < n; ++j) {
            if (matrix[0][j] == 0) rowFlag = true;
        }

        // if an element of matrix expect the first row and column is zero,
        // set corresponding indices in the first row and column to zero
```

```
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }

        // set corresponding element in matrix to zero
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }

        // set the first row and column to zero based on the flags
        if (colFlag == true) {
            for (int i = 0; i < m; ++i) matrix[i][0] = 0;
        }
        if (rowFlag == true) {
            for (int j = 0; j < n; ++j) matrix[0][j] = 0;
        }
    }
};
```

5.40 Spiral Matrix (M)

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].

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> res;
        if (matrix.empty()) return res;
        int rowBegin = 0, rowEnd = matrix.size()-1;
        int colBegin = 0, colEnd = matrix[0].size()-1;
```

```
        // Use four variables to locate search ranges and update values
        // based on the search progress
        while (rowBegin <= rowEnd && colBegin <= colEnd) {
            // Traverse Right
            for (int j = colBegin; j <= colEnd; ++j) {
                res.push_back(matrix[rowBegin][j]);
            }
            ++rowBegin;

            // Traverse Down
            for (int i = rowBegin; i <= rowEnd; ++i) {
                res.push_back(matrix[i][colEnd]);
            }
            --colEnd;

            // Traverse Left
            if (rowBegin <= rowEnd) {
                for (int j = colEnd; j >= colBegin; --j) {
                    res.push_back(matrix[rowEnd][j]);
                }
                --rowEnd;
            }

            // Traverse Up
            if (colBegin <= colEnd) {
                for (int i = rowEnd; i >= rowBegin; --i) {
                    res.push_back(matrix[i][colBegin]);
                }
                ++colBegin;
            }
        }

        return res;
    }
};
```

5.41 Spiral Matrix II (M)

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>> res(n, vector<int> (n));
        int start = 0, end = n-1, num = 1;
        // Iterate the matrix from outside level to insider level
        // Iterate each row and col from index 0 to n-2
        while (start < end) {
            for (int j = start; j < end; ++j) res[start][j] = num++;
            for (int i = start; i < end; ++i) res[i][end] = num++;
            for (int j = end; j > start; --j) res[end][j] = num++;
            for (int i = end; i > start; --i) res[i][start] = num++;
            ++start; // update index for the next inside level
            --end;
        }
        // In case that matrix[n/2][n/2] is the last num
        if (start == end) res[start][end] = num;
        return res;
    }
};
```

5.42 Rotate Image (M)

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?

```
// 1. Direct rotate
class Solution {
public:
    void rotate(vector<vector<int> > &matrix) {
        int n = matrix.size();
        for (int i = 0; i < n / 2; ++i) { // iterate layer by layer
            for (int j = i; j < n - 1 - i; ++j) {
                int tmp = matrix[i][j]; // upper left to tmp
                matrix[i][j] = matrix[n - 1 - j][i]; // bottom left to upper left
                matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j]; // bottom
                    right to bottom left
                matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i]; // upper
                    right to bottom right
                matrix[j][n - 1 - i] = tmp; // tmp to upper right
            }
        }
    }
};
```

```cpp
// 2. Flip by antidiagonal line, then flip by the middle row
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        // swap elements by antidiagonal line
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n-i; ++j)
                swap(matrix[i][j], matrix[n-1-j][n-1-i]);
        // swap elements by horizontal center line
        for (int i = 0; i < n/2; ++i)
            for (int j = 0; j < n; ++j)
                swap(matrix[i][j], matrix[n-1-i][j]);
    }
};

// 3. Transpose the matrix (i.e. flip by diagonal line), then reverse each row
class Solution {
public:
    void rotate(vector<vector<int> > &matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                swap(matrix[i][j], matrix[j][i]);
            }
            reverse(matrix[i].begin(), matrix[i].end());
        }
    }
};
```

Chapter 6

String

6.1 Add Binary (E)

Given two binary strings, return their sum (also a binary string).

For example, a = “11”, b = “1”, Return “100”.

```
class Solution {
public:
    string addBinary(string a, string b) {
        string res = "";
        int ai, bi, val, carry = 0;
        int i = a.size() - 1;
        int j = b.size() - 1;

        while (i >= 0 || j >= 0 || carry == 1) {
            if (i >= 0)
                ai = a[i--] - '0';          // convert char to integer
            else
                ai = 0;

            if (j >= 0)
                bi = b[j--] - '0';
            else
                bi = 0;

            val = (ai + bi + carry) % 2;
            carry = (ai + bi + carry) / 2;

            res = char(val + '0') + res;    // convert integer to char by
                char(val + '0')

        }
```

```
        return res;
    }
};
```

6.2 Count and Say (E)

The count-and-say sequence is the sequence of integers beginning as follows: 1, 11, 21, 1211, 111221, ...
1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.

Given an integer n, generate the nth sequence. The sequence of integers will be represented as a string.

```
class Solution {
public:
    string countAndSay(int n) {
        if (n == 0) return NULL;

        string res = "1";     // start at 1

        while (n != 1) {
            string cur = "";
            for (int i = 0; i < res.size(); ++i) {
                int cnt = 1;
                while ( (i+1) < res.size() && res[i] == res[i+1] ) {  // count
                    res[i]
                    ++cnt;
                    ++i;
                }
                cur += to_string(cnt) + res[i];                       // count and
                    say res[i]
            }
            res = cur;     // update current result
            --n;           // decrease n until n = 1
        }

        return res;
    }
};
```

6.3 Length of Last Word (E)

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string. If the last word does not exist, return 0. A word is defined as a character sequence consists of non-space characters only.

```
class Solution {
public:
    int lengthOfLastWord(string s) {
        if (s.empty()) return 0;

        int cnt = 0, n = s.size() - 1;

        while (s[n] == ' ' && n >= 0) {
            --n;
        }

        while (s[n] != ' ' && n >= 0) {
            ++cnt;
            --n;
        }

        return cnt;
    }
};
```

6.4 Longest Common Prefix (E)

Write a function to find the longest common prefix string amongst an array of strings.

```
class Solution {
public:
    string longestCommonPrefix(vector<string> &strs) {
        string prefix = "";                    // define the initial prefix
        if (strs.size() == 0) return prefix;

        int i, j;
        // iterate different chars in the first string
        for (i = 0; i < strs[0].size(); ++i) {

            // iterate different strings from the second string
            for (j = 1; j < strs.size() && i < strs[j].size(); ++j) {
                if (strs[j][i] != strs[0][i])    // compare each char between the
                    first string and the other strings
```

```
                return prefix;              // if no match for the current
                    char, return prefix immediately
            }

            if (j == strs.size())           // if the i-th char in the first
                string matches all others strings
                prefix += strs[0][i];       // update prefix
        }
        return prefix;
    }
};
```

6.5 Implement strStr() (E)

Implement strStr(). Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

```
class Solution {
public:
    int strStr(string haystack, string needle) {
        if (needle.empty()) return 0;

        for (int i = 0; ; ++i) {
            for (int j = 0; ; ++j) {
                if (needle[j] == '\0')      // if the current j reaches the end
                    of needle, i is the correct index
                    return i;
                if (haystack[i + j] == '\0') // if i+j reaches the end of
                    haystack, there is no chance to find the needle
                    return -1;
                if (haystack[i + j] != needle[j]) // break for each no-match
                    break;
            }
        }
    }
};
```

6.6 ZigZag Conversion (E)

Write the code that will take a string and make this conversion given a number of rows: string convert(string text, int nRows).
convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

```
class Solution {
public:
    string convert(string s, int numRows) {
        if (numRows <= 1) return s;

        vector<string> tmp(numRows);
        string res;
        int row = 0, flag = 1;

        for (int i = 0; i < s.size(); ++i) {
            tmp[row].push_back(s[i]);

            // perform zigzag
            if (row == 0)                // increase row number
                flag = 1;
            else if (row == numRows - 1) // decrease row number
                flag = -1;
            row += flag;
        }

        for (int i = 0; i < numRows; ++i)
            res.append(tmp[i]);

        return res;
    }
};
```

6.7 Group Shifted Strings (E)

Given a string, we can "shift" each of its letter to its successive letter, for example: "*abc*" $->$ "*bcd*". We can keep "shifting" which forms the sequence: "*abc*" $->$ "*bcd*" $->$ ... $->$ "*xyz*"

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.
For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],
Return: [["abc","bcd","xyz"], ["az","ba"], ["acef"], ["a","z"]]

Note: For the return value, each inner list's elements must follow the lexicographic order.

```
class Solution {
public:
    vector<vector<string>> groupStrings(vector<string>& strings) {
        vector<vector<string> > res;
        unordered_map<string, multiset<string>> m;
```

```
        for (auto a : strings) {
            string t = "";
            for (char c : a) {
                t += to_string((c + 26 - a[0]) % 26) + ",";
            }
            m[t].insert(a);
        }
        for (auto it = m.begin(); it != m.end(); ++it) {
            res.push_back(vector<string>(it->second.begin(), it->second.end()));
        }
        return res;
    }
};
```

6.8 Compare Version Numbers (E)

Compare two version numbers version1 and version2. If version1 > version2 return 1, if version1 < version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character. The . character does not represent a decimal point and is used to separate number sequences. For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering: $0.1 < 1.1 < 1.2 < 13.37$

```
class Solution {
public:
    int compareVersion(string version1, string version2) {
        int n1 = version1.size(), n2 = version2.size();
        int i, j, num1, num2;
        i = j = num1 = num2 = 0;

        //first compare the left part before ".", then compare the right part
            after "."
        while (i < n1 || j < n2) {
            while (i < n1 && version1[i] != '.')
                num1 = num1 * 10 + (version1[i++] - '0'); // string to int

            while (j < n2 && version2[j] != '.')
                num2 = num2 * 10 + (version2[j++] - '0');

            if (num1 > num2)
                return 1;
```

```
            else if (num1 < num2)
                return -1;

            num1 = num2 = 0;
            ++i;
            ++j;
        }

        return 0;
    }
};
```

6.9 Excel Sheet Column Number (E)

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:
$A -> 1$
$B -> 2$
$C -> 3$
...
$Z -> 26$
$AA -> 27$
$AB -> 28$

```
class Solution {
public:
    int titleToNumber(string s) {
        int result = 0;

        for (int i = 0; i < s.size(); ++i)
            result = result * 26 + (s[i] - 'A' + 1);

        return result;
    }
};
```

6.10 Excel Sheet Column Title (E)

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

$1 -> A$
$2 -> B$
$3 -> C$
...
$26 -> Z$
$27 -> AA$
$28 -> AB$

```
class Solution {
public:
    string convertToTitle(int n) {
        if (n <= 0)
            return NULL;

        string result;
        char tmp;

        while (n != 0) {
            n -= 1;
            tmp = n % 26 + 'A';
            result = tmp + result;
            n /= 26;
        }

        return result;
    }
};
```

6.11 Roman to Integer (E)

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

```
class Solution {
public:
    int romanToInt(string s) {
        int n = s.size();
        if (n == 0)
            return 0;

        unordered_map<char, int> roman = { { 'I' , 1 },
                                           { 'V' , 5 },
                                           { 'X' , 10 },
```

```
                                        { 'L' , 50 },
                                        { 'C' , 100 },
                                        { 'D' , 500 },
                                        { 'M' , 1000 } };

        int num = 0;
        for (int i = 0; i < n-1; ++i) {
            if (roman[s[i]] < roman[s[i+1]])
                num -= roman[s[i]];
            else
                num += roman[s[i]];
        }

        num += roman[s[n-1]];

        return num;
    }
};
```

6.12 Integer to Roman (M)

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

```
class Solution {
public:
    string intToRoman(int num) {
        int value[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
        string symbol[] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX",
            "V", "IV", "I"};
        string result;
        int digit, idx = 0;

        while (num != 0) {
            digit = num / value[idx];
            num %= value[idx];
            while (digit != 0) {
                result += symbol[idx];
                --digit;
            }
            ++idx;
        }

        return result;
```

```
    }
};
```

6.13 String to Integer (atoi) (E)

Implement atoi to convert a string to an integer.

```
class Solution {
public:
    int myAtoi(string str) {
        int sign = 1, base = 0, i = 0;

        while (str[i] == ' ')        // ignor whitespace
            ++i;

        if (str[i] == '-') {         // get the sign
            sign = -1;
            ++i;
        } else if (str[i] == '+'){
            sign = 1;
            ++i;
        }

        while (str[i] >= '0' && str[i] <= '9') {
            // catch the max int or min int case
            if (base > INT_MAX / 10 || (base == INT_MAX / 10 && str[i] - '0' >
                7)) {
                if (sign == 1)
                    return INT_MAX;
                else
                    return INT_MIN;
            }

            base = 10 * base + (str[i++] - '0');
        }

        return base * sign;
    }
};
```

6.14 Reverse String (E)

Write a function that takes a string as input and returns the string reversed.

Example: Given s = "hello", return "olleh".

```
class Solution {
public:
    string reverseString(string s) {
        int start = 0, end = s.length()-1;
        char tmp;
        while (start < end) {
            tmp = s[start];
            s[start++] = s[end];
            s[end--] = tmp;
        }
        return s;
    }
};
```

6.15 Reverse Vowels of a String (E)

Write a function that takes a string as input and reverse only the vowels of a string.

Example 1: Given s = "hello", return "holle".

Example 2: Given s = "leetcode", return "leotcede".

```
class Solution {
public:
    bool isVowel(char c) {
        return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c ==
            'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U';
    }

    string reverseVowels(string s) {
        int i = 0, j = s.length() - 1;

        while (i < j) {
            while (!isVowel(s[i]))
                ++i;
            while (!isVowel(s[j]))
                --j;
            if (i < j) {
```

```
                swap(s[i], s[j]);
                ++i;
                --j;
            }
        }

        return s;
    }
};
```

6.16 Valid Palindrome (E)

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,
"A man, a plan, a canal: Panama" is a palindrome.
"race a car" is not a palindrome.

```
class Solution {
public:
    bool isPalindrome(string s) {

        int start = 0, end = s.size()-1;

        while (start < end) {
            if (isalnum(s[start]) == false)     // isalnum: checks whether c is
                an alphanumeric character
                ++start;
            else if (isalnum(s[end]) == false)
                --end;
            else if (tolower(s[start++]) != tolower(s[end--])) // tolower:
                Converts parameter c to its lowercase equivalent if c is an
                uppercase letter and has a lowercase equivalent
                return false;
        }

        return true;
    }
};
```

6.17 Valid Anagram (E)

Given two strings s and t, write a function to determine if t is an anagram of s.

For example,
s = “anagram”, t = “nagaram”, return true.
s = “rat”, t = “car”, return false.

Note:
You may assume the string contains only lowercase alphabets.

Follow up:
What if the inputs contain unicode characters? How would you adapt your solution to such case?

```
// unordered_map<Key,T>::iterator it;
// (*it).first;          // the key value (of type Key)
// (*it).second;         // the mapped value (of type T)
// (*it);                // the "element value" (of type pair<const Key,T>)

// Sol1: Hash table
class Solution {
public:
    bool isAnagram(string s, string t) {
        if(s.length() != t.length())
            return false;

        unordered_map<char,int> counts;
        for (int i = 0; i < s.length(); ++i) {
            ++counts[s[i]];
            --counts[t[i]];
        }
        for (auto count : counts) {
            if (count.second)
                return false;
        }
        return true;
    }
};

Sol2: Optimization by the fix-size array
class Solution {
public:
     bool isAnagram(string s, string t) {
         if(s.length() != t.length())
             return false;
```

```
        int counts[26] = {0};
        for (int i = 0; i < s.length(); ++i) {
            ++counts[s[i] - 'a'];
            --counts[t[i] - 'a'];
        }
        for (int i = 0; i < 26; ++i) {
            if (counts[i])
                return false;
        }
        return true;
    }
};
```

6.18 Group Anagrams (M)

Given an array of strings, group anagrams together.

For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],
Return: [["ate", "eat","tea"], ["nat","tan"], ["bat"]].

Note: All inputs will be in lower-case.

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> group;

        for (string s : strs) {
            string key = s;
            sort(key.begin(), key.end());
            group[key].push_back(s);
        }

        vector<vector<string>> anagrams;
        for (auto g : group)
            anagrams.push_back(g.second);

        return anagrams;
    }
};
```

6.19 Palindrome Permutation (E)

Given a string, determine if a permutation of the string could form a palindrome.

For example, "code" $->$ False, "aab" $->$ True, "carerac" $->$ True.

```
// 1. Hashtable
class Solution {
public:
    bool canPermutePalindrome(string s) {
        unordered_map<char, int> m;
        int cnt = 0;

        for (auto a : s)
            ++m[a];

        for (auto it : m) {
            if (it.second % 2 == 1)
                ++cnt;
        }

        return cnt == 0 || ((cnt == 1) && (s.size() % 2 == 1));
    }
};

// 2. Set
class Solution {
public:
    bool canPermutePalindrome(string s) {
        set<char> t;
        for (auto a : s) {
            if (t.find(a) == t.end()) t.insert(a);
            else t.erase(a);
        }
        return t.empty() || t.size() == 1;
    }
};

// 3. Bitset
class Solution {
public:
    bool canPermutePalindrome(string s) {
        bitset<256> b;
        for (auto a : s) {
            b.flip(a);
        }
        return b.count() < 2;
```

```
    }
};
```

6.20 Palindrome Permutation II (M)

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:
Given s =“aabb”, return [“abba”, “baab”].
Given s = “abc”, return [].

```
class Solution {
public:
    vector<string> generatePalindromes(string s) {
        vector<string> res;
        unordered_map<char, int> m;
        string t = "", mid = "";

        for (auto a : s)
            ++m[a];

        for (auto it : m) {
            if (it.second % 2 == 1)
                mid += it.first;
            t += string(it.second / 2, it.first); // store the first half of
                string
            if (mid.size() > 1)                 // if the string is palindrome,
                mid size can only be 1
                return res;
        }

        permute(t, 0, mid, res);
        return res;
    }

    void permute(string &t, int start, string mid, vector<string> &res) {
        // palindrome permutation = permutation of the first half string
        //                        + the only char if exists
        //                        + reverse of the permutation of the first half
            string
        if (start >= t.size()) {
            res.push_back(t + mid + string(t.rbegin(), t.rend()));
        }
```

```
        // permutation
        for (int i = start; i < t.size(); ++i) {
            if (i != start && t[i] == t[start]) continue;
            swap(t[i], t[start]);
            permute(t, start + 1, mid, res);
            swap(t[i], t[start]);
        }
    }
};
```

6.21 Isomorphic Strings (E)

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t. All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,
Given "egg", "add", return true.
Given "foo", "bar", return false.
Given "paper", "title", return true.

```
class Solution {
public:
    bool isIsomorphic(string s, string t) {
        if (s.size() != t.size()) return false;

        int m1[128] = {0};
        int m2[128] = {0};
        for (int i = 0; i < s.size(); ++i) {
            if (m1[s[i]] != m2[t[i]])
                return false;
            m1[s[i]] = i+1;
            m2[t[i]] = i+1;
        }
        return true;
    }
};


// intuitive solution
class Solution {
public:
```

```
    bool isIsomorphic(string s, string t) {
        if (s.size()== 0) return true;

        unordered_map<char, char> map1;
        unordered_map<char, char> map2;
        map1.insert(make_pair(s[0], t[0]));
        map2.insert(make_pair(t[0], s[0]));

        for (int i = 1; i< s.size(); i++){
            if (map1.find(s[i])!=map1.end()){
                if (t[i] != map1.at(s[i])) {
                    return false;
                }
            }

            if (map2.find(t[i])!=map2.end()){

                if (s[i] != map2.at(t[i])) {
                    return false;
                }
            }

            else {
                map1.insert(make_pair(s[i],t[i]));
                map2.insert(make_pair(t[i],s[i]));
            }
        }
        return true;
    }
};
```

6.22 Word Pattern (E)

Given a pattern and a string str, find if str follows the same pattern. Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.

Examples:
pattern = “abba”, str = “dog cat cat dog” should return true.
pattern = “abba”, str = “dog cat cat fish” should return false.
pattern = “aaaa”, str = “dog cat cat dog” should return false.
pattern = “abba”, str = “dog dog dog dog” should return false.

```
class Solution {
public:
    bool wordPattern(string pattern, string str) {
```

```
        unordered_map<char, int> p;
        unordered_map<string, int> w;
        istringstream in(str);          // read each word from str to in
        int i = 0;

        for (string word; in >> word; ++i) {
            if (p.find(pattern[i]) != p.end() || w.find(word) != w.end()) {
                if (p[pattern[i]] != w[word]) return false;
            } else {
                p[pattern[i]] = w[word] = i + 1;
            }
        }
        return i == pattern.size();
    }
};
```

6.23 Word Pattern II (H)

Given a pattern and a string str, find if str follows the same pattern. Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty substring in str.

Examples:
pattern = "abab", str = "redblueredblue" should return true.
pattern = "aaaa", str = "asdasdasdasd" should return true.
pattern = "aabb", str = "xyzabcxzyabc" should return false.

Notes: You may assume both pattern and str contains only lowercase letters.

```
class Solution {
public:
    bool wordPatternMatch(string pattern, string str) {
        unordered_map<char, string> m;
        set<string> s;
        return helper(pattern, 0, str, 0, m, s);
    }
    bool helper(string pattern, int p, string str, int r, unordered_map<char,
        string> &m, set<string> &s) {
        if (p == pattern.size() && r == str.size()) return true;
        if (p == pattern.size() || r == str.size()) return false;
        char c = pattern[p];
        for (int i = r; i < str.size(); ++i) {
            string t = str.substr(r, i - r + 1);
            if (m.count(c) && m[c] == t) {
```

```
                if (helper(pattern, p + 1, str, i + 1, m, s)) return true;
            } else if (!m.count(c)) {
                if (s.count(t)) continue;
                m[c] = t;
                s.insert(t);
                if (helper(pattern, p + 1, str, i + 1, m, s)) return true;
                m.erase(c);
                s.erase(t);
            }
        }
        return false;
    }
};
```

6.24 Valid Parentheses (E)

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> tmp;

        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(' || s[i] == '[' || s[i] == '{') {
                tmp.push(s[i]);
            } else if ( (!tmp.empty() && tmp.top() == '(' && s[i] == ')') ||
                        (!tmp.empty() && tmp.top() == '[' && s[i] == ']') ||
                        (!tmp.empty() && tmp.top() == '{' && s[i] == '}') ){
                tmp.pop();
            } else {
                return false;
            }
        }

        return tmp.empty();
    }
};
```

6.25 Generate Parentheses (M)

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> res;
        generate(res, "", n, 0);
        return res;
    }

    void generate(vector<string> &res, string str, int left, int right) {
        if (left == 0 && right == 0) {
            res.push_back(str);
            return;
        }

        if (right > 0)
            generate(res, str+")", left, right-1);

        if (left > 0)
            generate(res, str+"(", left-1, right+1);
    }
};
```

6.26 Different Ways to Add Parentheses (M)

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and *.

Example: Input: "2-1-1".
((2-1)-1) = 0
(2-(1-1)) = 2
Output: [0, 2]

```
class Solution {
public:
    vector<int> diffWaysToCompute(string input) {
        vector<int> res;
        int n = input.size();

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

```
            char c = input[i];
            if (c == '+' || c == '-' || c == '*') {
                vector<int> result1 = diffWaysToCompute(input.substr(0,i));
                vector<int> result2 = diffWaysToCompute(input.substr(i+1));
                for (int j = 0; j < result1.size(); ++j) {
                    for (int k = 0; k < result2.size(); ++k) {
                        switch (c) {
                            case '+':
                                res.push_back(result1[j] + result2[k]);
                                break;
                            case '-':
                                res.push_back(result1[j] - result2[k]);
                                break;
                            case '*':
                                res.push_back(result1[j] * result2[k]);
                                break;
                        }
                    }
                }
            }
        }

        if (res.empty())
            res.push_back(stoi(input));

        return res;
    }
};
```

6.27 Longest Valid Parentheses (H)

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.

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int maxlen = 0, last = -1;
        stack<int> tmp;
```

```
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(') {
                tmp.push(i);              // push the index of '(' to stack
            } else {
                if (tmp.empty()) {        // if no match for the current ')'
                    last = i;             // update last index of '('
                } else {                  // if find a match for the current ')'
                    tmp.pop();            // pop the index of '('
                    if (tmp.empty()) {
                        maxlen = max(maxlen, i - last);
                    } else {
                        maxlen = max(maxlen, i - tmp.top());
                    }
                }
            }
        }

        return maxlen;
    }
};
```

6.28 Remove Invalid Parentheses (H)

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

```
class Solution {
public:
    vector<string> removeInvalidParentheses(string s) {
        unordered_set<string> result;
        int left_removed = 0;
        int right_removed = 0;
        for(auto c : s) {
            if(c == '(') {
                ++left_removed;
            }
            if(c == ')') {
                if(left_removed != 0) {
                    --left_removed;
                }
                else {
                    ++right_removed;
                }
```

```
            }
        }
        helper(s, 0, left_removed, right_removed, 0, "", result);
        return vector<string>(result.begin(), result.end());
    }
private:
    void helper(string s, int index, int left_removed, int right_removed, int
        pair, string path, unordered_set<string>& result) {
        if(index == s.size()) {
            if(left_removed == 0 && right_removed == 0 && pair == 0) {
                result.insert(path);
            }
            return;
        }
        if(s[index] != '(' && s[index] != ')') {
            helper(s, index + 1, left_removed, right_removed, pair, path +
                s[index], result);
        }
        else {
            if(s[index] == '(') {
                if(left_removed > 0) {
                    helper(s, index + 1, left_removed - 1, right_removed, pair,
                        path, result);
                }
                helper(s, index + 1, left_removed, right_removed, pair + 1, path +
                    s[index], result);
            }
            if(s[index] == ')') {
                if(right_removed > 0) {
                    helper(s, index + 1, left_removed, right_removed - 1, pair,
                        path, result);
                }
                if(pair > 0) {
                    helper(s, index + 1, left_removed, right_removed, pair - 1,
                        path + s[index], result);
                }
            }
        }
    }
};
```

6.29 Flip Game (E)

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "- -". The game ends when a person can no longer make a move and therefore the other

person will be the winner. Write a function to compute all possible states of the string after one valid move.

For example, given s = “++++”, after one move, it may become one of the following states: [“- -++”, “+- -+”, “++- -”] If there is no valid move, return an empty list [].

```
class Solution {
public:
    vector<string> generatePossibleNextMoves(string s) {
        vector<string> res;
        for (int i = 1; i < s.size(); ++i) {
            if (s[i] == '+' && s[i-1] == '+'){
                res.push_back(s.substr(0, i-1) + "--" + s.substr(i+1)); //
                    substr(pos, len);
            }
        }
        return res;
    }
};
```

6.30 Flip Game II (M)

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive ”++” into ”–”. The game ends when a person can no longer make a move and therefore the other person will be the winner. Write a function to determine if the starting player can guarantee a win.

For example, given s = ”++++”, return true. The starting player can guarantee a win by flipping the middle ”++” to become ”+–+”.

```
class Solution {
public:
    bool canWin(string s) {
        for (int i = 1; i < s.size(); ++i) {
            if (s[i] == '+' && s[i - 1] == '+' && !canWin(s.substr(0, i - 1) +
                "--" + s.substr(i + 1))) {
                return true;
            }
        }
        return false;
    }
};
```

6.31 Shortest Word Distance (E)

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example,
Assume that words = [“practice”, “makes”, “perfect”, “coding”, “makes”].
Given word1 = “coding”, word2 = “practice”, return 3.
Given word1 = “makes”, word2 = “coding”, return 1.

Note: You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

```
class Solution {
public:
    int shortestDistance(vector<string>& words, string word1, string word2) {
        int p1 = -1, p2 = -1, res = INT_MAX;
        for (int i = 0; i < words.size(); ++i) {
            if (words[i] == word1) p1 = i;
            if (words[i] == word2) p2 = i;
            if (p1 != -1 && p2 != -1)
                res = min(res, abs(p1 - p2));
        }
        return res;
    }
};

class Solution {
public:
    int shortestDistance(vector<string>& words, string word1, string word2) {
        int idx = -1, res = INT_MAX;
        for (int i = 0; i < words.size(); ++i) {
            if (words[i] == word1 || words[i] == word2) {
                if (idx != -1 && words[idx] != words[i]) {
                    res = min(res, i - idx)
                }
                idx = i;
            }
        }
        return res;
    }
};
```

6.32 Shortest Word Distance II (M)

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it? Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example,
Assume that words = [“practice”, “makes”, “perfect”, “coding”, “makes”].
Given word1 = “coding”, word2 = “practice”, return 3.
Given word1 = “makes”, word2 = “coding”, return 1.

Note: You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

```
// 1. O(MN) solution
class WordDistance {
public:
    WordDistance(vector<string> &words) {
        for (int i = 0; i < words.size(); ++i) {
            m[words[i]].push_back(i);
        }
    }

    int shortestDistance(string word1, string word2) {
        int res = INT_MAX;
        for (int i = 0; i < m[word1].size(); ++i) {
            for (int j = 0; j < m[word2].size(); ++j) {
                res = min(res, abs(m[word1][i] - m[word2][j]));
            }
        }
        return res;
    }

private:
    unordered_map<string, vector<int>> m;
};

// 2. O(M+N) solution
class WordDistance {
public:
    WordDistance(vector<string> &words) {
        for (int i = 0; i < words.size(); ++i) {
            m[words[i]].push_back(i);
        }
```

```
    }

    int shortestDistance(string word1, string word2) {
        int i = 0, j = 0, res = INT_MAX;
        while (i < m[word1].size() && j < m[word2].size()) {
            res = min(res, abs(m[word1][i] - m[word2][j]));
            m[word1][i] < m[word2][j] ? ++i : ++j;
        }
        return res;
    }

private:
    unordered_map<string, vector<int>> m;
};
```

6.33 Shortest Word Distance III (M)

This is a follow up of Shortest Word Distance. The only difference is now word1 could be the same as word2. Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list. In this time, word1 and word2 may be the same and they represent two individual words in the list.
For example,
Assume that words = [“practice”, “makes”, “perfect”, “coding”, “makes”].
Given word1 = “makes”, word2 = “coding”, return 1.
Given word1 = “makes”, word2 = “makes”, return 3.

Note: You may assume word1 and word2 are both in the list.

```
class Solution {
public:
    int shortestWordDistance(vector<string>& words, string word1, string word2) {
        int idx = -1, res = INT_MAX;
        for (int i = 0; i < words.size(); ++i) {
            if (words[i] == word1 || words[i] == word2) {
                if (idx != -1 && (word1 == word2 || words[i] != words[idx])) {
                    res = min(res, i - idx);
                }
                idx = i;
            }
        }
        return res;
    }
};
```

6.34 Strobogrammatic Number (E)

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down). Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers "69", "88", and "818" are all strobogrammatic.

```
class Solution{
public:
    bool isStrobogrammatic(string num) {
        int l = 0, r = num.size() - 1;
        while (l <= r) {
            if (num[l] == num[r]) {
                if (num[l] != '1' || num[l] != '8' || num[l] != '0') {
                    return false;
                }
            } else {
                if ( (num[l] != '6' || num[r] != '9') && (num[l] != '9' || num[r]
                    != '6') ) {
                    return false;
                }
            }
            ++l;
            --r;
        }
        return true;
    }
};
```

6.35 Strobogrammatic Number II (M)

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down). Find all strobogrammatic numbers that are of length = n.

For example, Given n = 2, return ["11","69","88","96"].

Hint: Try to use recursion and notice that it should recurse with n - 2 instead of n - 1.

```
class Solution {
public:
    vector<string> findStrobogrammatic(int n) {
        return find(n, n);
    }
```

```
    vector<string> find(int m, int n) {
        if (m == 0) return {""};
        if (m == 1) return {"0", "1", "8"};
        vector<string> t = find(m - 2, n), res;

        for (auto a : t) {
            // add 0 to both side of a if level m is not level n
            if (m != n) res.push_back("0" + a + "0");

            res.push_back("1" + a + "1");
            res.push_back("6" + a + "9");
            res.push_back("8" + a + "8");
            res.push_back("9" + a + "6");
        }

        return res;
    }
};
```

6.36 Strobogrammatic Number III (M)

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down). Write a function to count the total strobogrammatic numbers that exist in the range of $low \leq num \leq high$.

For example,
Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.

Note: Because the range might be a large number, the low and high numbers are represented as string.

```
class Solution {
public:
    int strobogrammaticInRange(string low, string high) {
        int res = 0;
        find(low, high, "", res);
        find(low, high, "0", res);
        find(low, high, "1", res);
        find(low, high, "8", res);
        return res;
    }
    void find(string low, string high, string w, int &res) {
```

```
        if (w.size() >= low.size() && w.size() <= high.size()) {
            if ((w.size() == low.size() && w.compare(low) < 0) || (w.size() ==
                high.size() && w.compare(high) > 0)) {
                return;
            }
            if (!(w.size() > 1 && w[0] == '0')) ++res;
        }
        if (w.size() + 2 > high.size()) return;
        find(low, high, "0" + w + "0", res);
        find(low, high, "1" + w + "1", res);
        find(low, high, "6" + w + "9", res);
        find(low, high, "8" + w + "8", res);
        find(low, high, "9" + w + "6", res);
    }
};
```

6.37 Read N Characters Given Read4 (E)

The API: int read4(char *buf) reads 4 characters at a time from a file. The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file. By using the read4 API, implement the function int read(char *buf, int n) that reads n characters from the file.

Note:The read function will only be called once for each test case.

```
int read4(char *buf);

class Solution {
public:
    int read(char *buf, int n) {
        int res = 0;
        for (int i = 0; i <= n / 4; ++i) {
            int cur = read4(buf + res);
            if (cur == 0) break;
            res += cur;
        }
        return min(res, n);
    }
};
```

6.38 Read N Characters Given Read4 II (H)

The API: int read4(char *buf) reads 4 characters at a time from a file. The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file. By using the read4 API, implement the function int read(char *buf, int n) that reads n characters from the file.

Note: The read function may be called multiple times.

```
class Solution {
public:
    int read(char *buf, int n) {
        for (int i = 0; i < n; ++i) {
            if (readPos == writePos) {
                writePos = read4(buff);
                readPos = 0;
                if (writePos == 0) return i;
            }
            buf[i] = buff[readPos++];
        }
        return n;
    }
private:
    int readPos = 0, writePos = 0;
    char buff[4];
};
```

6.39 Unique Word Abbreviation (E)

An abbreviation of a word follows the form $< firstletter >< number >< lastletter >$. Below are some examples of word abbreviations:

a) it $--> $ it (no abbreviation)

b) d—o—g $--> $ d1g

c) i—nternationalizatio—n $--> $ i18n

d) l—ocalizatio—n $--> $ l10n

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation.

Example:
Given dictionary = ["deer", "door", "cake", "card"]
isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true

```
class ValidWordAbbr {
public:
    ValidWordAbbr(vector<string> &dictionary) {
        for (auto a : dictionary) {
            // get the word abbreviation in dictionary
            string k = a.front() + to_string(a.size() - 2) + a.back();
            m[k].insert(a);
        }
    }

    // If this word (also this word's abbreviation) is not in the dictionary
    // OR this word and only it's abbreviation in the dictionary,
    // we call a word's abbreviation unique.
    bool isUnique(string word) {
        // get the word abbreviation
        string k = word.front() + to_string(word.size() - 2) + word.back();
        return m[k].count(word) == m[k].size();
    }

private:
    unordered_map<string, set<string>> m; // mapping the word abbr and the words
};
```

6.40 Generalized Abbreviation (M)

Write a function to generate the generalized abbreviations of a word.

Example: Given word = "word", return the following list (order does not matter):
["word","1ord","w1rd","wo1d","wor1","2rd","w2d","wo2","1o1d","1or1","w1r1",
"1o2","2r1","3d","w3","4"]

```
// 1. Iterative solution
class Solution {
public:
    vector<string> generateAbbreviations(string word) {
        vector<string> res;
        // Use binary bit to code word
```

```
        for (int i = 0; i < pow(2, word.size()); ++i) {
            string out = "";
            int cnt = 0, t = i;
            for (int j = 0; j < word.size(); ++j) {
                if (t & 1 == 1) { // If bit is 1, count it as a number
                    ++cnt;
                    if (j == word.size() - 1) {
                        out += to_string(cnt);
                    }
                } else { // If bit is 0, count it as a char
                    if (cnt != 0) { // If there is a number counted, output it
                        out += to_string(cnt);
                        cnt = 0;
                    }
                    out += word[j]; // Otherwise output char
                }
                t >>= 1; // Move to next bit
            }
            res.push_back(out);
        }
        return res;
    }
};

// 2. Recursive solution
class Solution {
public:
    vector<string> generateAbbreviations(string word) {
        vector<string> res{word};
        helper(word, 0, res);
        return res;
    }
    void helper(string word, int pos, vector<string> &res) {
        for (int i = pos; i < word.size(); ++i) {
            for (int j = 1; i + j <= word.size(); ++j) {
                string t = word.substr(0, i);
                t += to_string(j) + word.substr(i + j);
                res.push_back(t);
                helper(t, i + 1 + to_string(j).size(), res);
            }
        }
    }
};
```

Chapter 7

Tree

7.1 Binary Tree Traversal

7.1.1 Binary Tree Preorder Traversal (M)

Given a binary tree, return the preorder traversal of its nodes' values.

For example: Given binary tree [1,null,2,3], return [1,2,3].

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
// 1. Recursion solution
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> res;
        preorder(root, res);
        return res;
    }
    void preorder(TreeNode *root, vector<int> &res) {
        if (!root) return;
        res.push_back(root->val);
        if (root->left) preorder(root->left, res);
        if (root->right) preorder(root->right, res);
    }
};
```

```
// 2. Non-recursion
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> res;
        stack<TreeNode*> s;
        if (!root) return res;
        s.push(root);
        while (!s.empty()) {
            TreeNode *p = s.top();
            s.pop();
            res.push_back(p->val);
            if (p->right) s.push(p->right);
            if (p->left) s.push(p->left);
        }
        return res;
    }
};
```

7.1.2 Binary Tree Inorder Traversal (M)

Given a binary tree, return the inorder traversal of its nodes' values.

For example: Given binary tree [1,null,2,3], return [1,3,2].

```
// 1. Recursion solution
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> res;
        inorder(root, res);
        return res;
    }
    void inorder(TreeNode *root, vector<int> &res) {
        if (!root) return;
        if (root->left) inorder(root->left, res);
        res.push_back(root->val);
        if (root->right) inorder(root->right, res);
    }
};

// 2. Non-recursion
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root){
        vector<int> res;
```

```
        stack<TreeNode*> s;
        TreeNode *p = root;
        while (p || !s.empty()) {
            if (p) {
                s.push(p);
                p = p->left;
            } else {
                p = s.top();
                s.pop();
                res.push_back(p->val);
                p = p->right;
            }
        }
        return res;
    }
};
```

7.1.3 Binary Tree Postorder Traversal (H)

Given a binary tree, return the postorder traversal of its nodes' values.

For example: Given binary tree [1,null,2,3], return [3,2,1].

```
// 1. Recursion solution
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> res;
        postorder(root, res);
        return res;
    }
    void postorder(TreeNode *root, vector<int> &res) {
        if (!root) return;
        if (root->left) postorder(root->left, res);
        if (root->right) postorder(root->right, res);
        res.push_back(root->val);
    }
};

// 2. Non-recursion
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> res;
        if (!root) return res;
        stack<TreeNode*> s;
```

```
        s.push(root);
        TreeNode *p = root;
        while (!s.empty()) {
            TreeNode *top = s.top();
            // If the current node top in stack doesn't have any child in tree,
            // or its left or right child has been visited,
            // push back the current element to res
            if ((!top->left && !top->right) || top->left == p || top->right == p)
                {
                res.push_back(top->val);
                s.pop();
                p = top;
            } else { // Otherwise, push its right and left child to stack
                if(top->right) s.push(top->right);
                if(top->left) s.push(top->left);
            }
        }
        return res;
    }
};
```

7.1.4 Binary Tree Level Order Traversal (E)

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        BFS(root, result, 0);
        return result;
    }

    void BFS(TreeNode *root, vector<vector<int>> &res, int depth) {
        if (root == NULL)
            return;

        if (res.size() == depth)         // The level does not exist in output
            res.push_back(vector<int>()); // Create a new level

        res[depth].push_back(root->val); // Add the current value to its level

        BFS(root->left, res, depth+1);   // Go to the next level
        BFS(root->right, res, depth+1);
    }
```

```
};
```

7.1.5 Binary Tree Level Order Traversal II (E)

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).

```
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> result;
        BFS(root, result, 0);
        reverse(result.begin(), result.end());     // reverse to get the
            bottom-up level order result
        return result;
    }

    void BFS(TreeNode *root, vector<vector<int>> &res, int depth) {
        if (root == NULL)
            return;

        if (res.size() == depth)         // The level does not exist in output
            res.push_back(vector<int>()); // Create a new level

        res[depth].push_back(root->val); // Add the current value to its level
        BFS(root->left, res, depth+1);   // Go to the next level
        BFS(root->right, res, depth+1);
    }
};
```

7.1.6 Binary Tree Zigzag Level Order Traversal (M)

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).

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> res;
        BFS(root, res, 0);
        return res;
    }

    void BFS(TreeNode *root, vector<vector<int>> &res, int depth) {
```

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

        if (res.size() == depth)
            res.push_back(vector<int>());

        if (depth % 2 == 0) {
            res[depth].push_back(root->val);
        } else {
            res[depth].insert(res[depth].begin(), root->val); // insert the
                current root->val to the beginning of vector
        }

        BFS(root->left, res, depth+1);
        BFS(root->right, res, depth+1);
    }
};
```

7.1.7 Binary Tree Right Side View (M)

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].

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        rightSideView(root, res, 1);
        return res;
    }

    void rightSideView(TreeNode *p, vector<int> &res, int level) {
        if (p == NULL) return;
        // push back p->val onlf if it is the right most node in the current level
        if (res.size() < level) res.push_back(p->val);
        rightSideView(p->right, res, level+1);
        rightSideView(p->left, res, level+1);
    }
};
```

7.1.8 Populating Next Right Pointers in Each Node (M)

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).

```
/**
 * 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) {}
 * };
 */
// 1. Recursion, more than constant space
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (!root) return;
        if (root->left) root->left->next = root->right;
        if (root->right) root->right->next = root->next? root->next->left : NULL;
        connect(root->left);
        connect(root->right);
    }
};

// 2. Non-recursion, more than constant space
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (!root) return;
        queue<TreeLinkNode*> q;
        q.push(root);
        q.push(NULL);
        while (true) {
            TreeLinkNode *cur = q.front();
```

```
            q.pop();
            if (cur) {
                cur->next = q.front();
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            } else {
                if (q.size() == 0 || q.front() == NULL) return;
                q.push(NULL);
            }
        }
    }
};

// 3. Non-recursion, constant space
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (!root) return;
        while (root->left) {
            TreeLinkNode *p = root;
            while(p) { //level order travesal from left to right
                p->left->next = p->right;
                if (p->next) p->right->next = p->next->left;
                p = p->next;
            }
            root = root->left;
        }
    }
};
```

7.1.9 Populating Next Right Pointers in Each Node II (H)

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.

```
// 1. Recursion, more than constant space
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (!root) return;
        TreeLinkNode *p = root->next;
        while (p) {
            if (p->left) {
```

```
                p = p->left;
                break;
            }
            if (p->right) {
                p = p->right;
                break;
            }
            p = p->next;
        }
        if (root->right) root->right->next = p;
        if (root->left) root->left->next = root->right ? root->right : p;
        connect(root->right);
        connect(root->left);
    }
};

// 2. Non-recursion, more than constant space
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (!root) return;
        queue<TreeLinkNode*> q;
        q.push(root);
        q.push(NULL);
        while (true) {
            TreeLinkNode *cur = q.front();
            q.pop();
            if (cur) {
                cur->next = q.front();
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            } else {
                if (q.size() == 0 || q.front() == NULL) return;
                q.push(NULL);
            }
        }
    }
};

// 3. Non-recursion, constant space
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (!root) return;
        TreeLinkNode *leftMost = root;
        while (leftMost) {
            TreeLinkNode *p = leftMost;
```

```
            while (p && !p->left && !p->right) p = p->next;
            if (!p) return;
            leftMost = p->left ? p->left : p->right;
            TreeLinkNode *cur = leftMost;
            while (p) {
                if (cur == p->left) {
                    if (p->right) {
                        cur->next = p->right;
                        cur = cur->next;
                    }
                    p = p->next;
                }
                else if (cur == p->right) {
                    p = p->next;
                } else {
                    if (!p->left && !p->right) {
                        p = p->next;
                        continue;
                    }
                    cur->next = p->left ? p->left : p->right;
                    cur = cur->next;
                }
            }
        }
    }
};
```

7.2 Binary Tree Recursion

7.2.1 Same Tree (E)

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

```
/**
 * 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 isSameTree(TreeNode* p, TreeNode* q) {
        if (p == NULL && q == NULL) return true;
        if (p == NULL || q == NULL) return false;
        return p->val == q->val &&
               isSameTree(p->left, q->left) &&
               isSameTree(p->right, q->right);
    }
};
```

7.2.2 Symmetric Tree (E)

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (!root) return true;
        return compare(root->left, root->right);
    }

    bool compare(TreeNode *p, TreeNode *q) {
        if (!p && !q)  return true;
        if (!p || !q)  return false;
        if (p->val != q->val)  return false;
        return compare(p->left, q->right) && compare(p->right, q->left);
    }
};
```

7.2.3 Invert Binary Tree (E)

Invert a binary tree 4-2-7-1-3-6-9 to 4-7-2-9-6-3-1.

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL)
            return NULL;

        TreeNode *tmp = root->left;
        root->left = invertTree(root->right);
        root->right = invertTree(tmp);
        return root;
```

```
    }
};
```

7.2.4 Binary Tree Upside Down (M)

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example: Given a binary tree 1,2,3,4,5, return the root of the binary tree [4,5,2,#,#,3,1].

```
class Solution {
public:
    TreeNode *upsideDownBinaryTree(TreeNode *root) {
        if (!root || !root->left) return root;
        TreeNode *l = root->left, *r = root->right;
        TreeNode *res = upsideDownBinaryTree(l);
        l->left = r;
        l->right = root;
        root->left = root_right = NULL;
        return res;
    }
};
```

7.2.5 Lowest Common Ancestor of a Binary Tree (M)

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == NULL || root == p || root == q)
            return root;

        TreeNode *left = lowestCommonAncestor(root->left, p, q);
        TreeNode *right = lowestCommonAncestor(root->right, p, q);

        if (left == NULL) {
            return right;
        } else {
            if (right == NULL) {
                return left;
            } else {
                return root;
```

```
            }
        }
    }
};
```

7.2.6 Binary Tree Longest Consecutive Sequence (M)

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,
$[1, NULL, 3, 2, 4, NULL, NULL, NULL, 5]$, Longest consecutive sequence path is 3-4-5, so return 3.
$[2, NULL, 3, 2, NULL, 1, NULL]$, Longest consecutive sequence path is 2-3, not 3-2-1, so return 2.

```
class Solution {
public:
    int longestConsecutive(TreeNode* root) {
        if (!root) return 0;
        int res = 0;
        dfs(root, 1, res);
        return res;
    }
    void dfs(TreeNode *root, int len, int &res) {
        res = max(res, len);
        if (root->left) {
            if (root->left->val == root->val + 1)
                dfs(root->left, len + 1, res);
            else
                dfs(root->left, 1, res);
        }
        if (root->right) {
            if (root->right->val == root->val + 1)
                dfs(root->right, len + 1, res);
            else
                dfs(root->right, 1, res);
        }
    }
};
```

7.2.7 Count Univalue Subtrees (M)

Given a binary tree, count the number of uni-value subtrees. A Uni-value subtree means all nodes of the subtree have the same value.

For example: Given binary tree, [5,1,5,5,5,#,5], return 4.

```
class Solution {
public:
    int countUnivalSubtrees(TreeNode* root) {
        if (!root) return res;
        if (isUnival(root, root->val)) ++res;
        countUnivalSubtrees(root->left);
        countUnivalSubtrees(root->right);
        return res;
    }
private:
    int res = 0;
    bool isUnival(TreeNode *root, int val) {
        if (!root) return true;
        return root->val == val && isUnival(root->left, val) &&
            isUnival(root->right, val);
    }
};
```

7.2.8 Balanced Binary Tree (E)

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.

```
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        if (root == NULL)
            return true;

        return isBalanced(root->left) && isBalanced(root->right) &&
            abs(height(root->left) - height(root->right)) <= 1;
    }

    int height(TreeNode *node) {
        if (node != NULL)
            return 1 + max(height(node->left), height(node->right));
```

```
        else
            return 0;
    }
};
```

7.2.9 Maximum Depth of Binary Tree (E)

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root != NULL)
            return 1 + max(maxDepth(root->left), maxDepth(root->right));
        else
            return 0;
    }
};
```

7.2.10 Minimum Depth of Binary Tree (E)

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.

```
class Solution {
public:
    int minDepth(TreeNode* root) {
        if (!root)
            return 0;
        if (!root->left)
            return 1 + minDepth(root->right);
        if (!root->right)
            return 1 + minDepth(root->left);

        return 1 + min(minDepth(root->left), minDepth(root->right));
    }
};
```

7.2.11 Find Leaves of Binary Tree (M)

Given a binary tree, find all leaves and then remove those leaves. Then repeat the previous steps until the tree is empty.

Example:
Given binary tree [1, 2, 3, 4, 5], Returns [4, 5, 3], [2], [1].

```
class Solution {
public:
    vector<vector<int>> findLeaves(TreeNode* root) {
        vector<vector<int>> res;
        helper(root, res);
        return res;
    }

    int helper(TreeNode *root, vector<vector<int>> &res) {
        if (!root) return -1;
        int depth = 1 + max(helper(root->left, res), helper(root->right, res));

        if (depth >= res.size()) {
                // assign new level
                // or we can use: res.push_back(vector<int>())
            res.resize(depth + 1);
        }

        res[depth].push_back(root->val);
        return depth;
    }
};
```

7.2.12 Binary Tree Paths (E)

Given a binary tree, return all root-to-leaf paths.

```
class Solution {
public:
    vector<string> binaryTreePaths(TreeNode *root) {
        vector<string> res;
        if (!root) return res;
        getTreePaths(root, res, to_string(root->val));
        return res;
    }

    void getTreePaths(TreeNode *root, vector<string> &res, string s) {
```

```
        if (!root->left && !root->right) {  // push back to res until the end of
            the leaf
            res.push_back(s);
            return;
        }

        if (root->left)
            getTreePaths(root->left, res, s + "->" + to_string(root->left->val));
        if (root->right)
            getTreePaths(root->right, res, s + "->" +
                to_string(root->right->val));
    }

};
```

7.2.13 Path Sum (E)

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

```
class Solution {
public:
    bool hasPathSum(TreeNode *root, int sum){
        if (!root) return false;

        if (!root->left && !root->right)
            return root->val == sum;

        return hasPathSum(root->left, sum-(root->val)) || hasPathSum(root->right,
            sum-(root->val));
    }
};
```

7.2.14 Path Sum II (M)

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

```
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode *root, int sum) {
        vector<vector<int>> res;
        vector<int> path;
        getPathSum(root, sum, res, path);
```

```
        return res;
    }

    void getPathSum(TreeNode *root, int sum, vector<vector<int>> &res,
        vector<int> &path) {
        if (!root) return;

        path.push_back(root->val);          // add the current node to the path
        if (!root->left && !root->right) {
            if (root->val == sum)
                res.push_back(path);        // find a correct path's sum
        }

        getPathSum(root->left, sum-(root->val), res, path);
        getPathSum(root->right, sum-(root->val), res, path);

        path.pop_back();                    // delete the current node if no
            correct path's sum has been found
    }
};
```

7.2.15 Sum Root to Leaf Numbers (M)

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path $1- > 2- > 3$ which represents the number 123. Find the total sum of all root-to-leaf numbers.

For example:
The root-to-leaf path $1- > 2$ represents the number 12. The root-to-leaf path $1- > 3$ represents the number 13. Return the sum $= 12 + 13 = 25$.

```
class Solution {
public:
    int sumNumbers(TreeNode *root) {
        return sumTreePaths(root, 0);
    }

    int sumTreePaths(TreeNode *root, int sum) {
        if (!root) return 0;

        sum = 10 * sum + root->val;         // e.g. 1->2: 1 * 10 + 2 = 12

        if (!root->left && !root->right) {
            return sum;
        } else {
```

```
            return sumTreePaths(root->left, sum) + sumTreePaths(root->right, sum);
        }
    }
};
```

7.2.16 Binary Tree Maximum Path Sum (H)

Given a binary tree, find the maximum path sum. For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

```
class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int sum = INT_MIN;
        maxSum(root, sum);
        return sum;
    }

    int maxSum(TreeNode *root, int &sum) {
        if (!root) return 0;

        int left = max(0, maxSum(root->left, sum));
        int right = max(0, maxSum(root->right, sum));
        sum = max(sum, left + right + root->val);  // total sum = root + left
            subtree + right subtree
        return max(left, right) + root->val;       // subtree sum = root +
            max(left subtree, right subtree)
    }
};
```

7.3 Binary Tree Construction

7.4 Binary Search Tree

7.4.1 Lowest Common Ancestor of a Binary Search Tree (E)

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants

(where we allow a node to be a descendant of itself)."

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (p->val < root->val && q->val < root->val) {
            return lowestCommonAncestor(root->left, p, q);
        } else if (p->val > root->val && q->val > root->val) {
            return lowestCommonAncestor(root->right, p, q);
        } else {
            return root;
        }
    }
};
```

7.4.2 Validate Binary Search Tree (M)

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.

```
// 1. Recursion without inorder traversal
class Solution {
public:
    bool isValidBST(TreeNode *root) {
        return isValidBST(root, LONG_MIN, LONG_MAX);
    }
    bool isValidBST(TreeNode *root, long mn, long mx) {
        if (!root) return true;
        if (root->val <= mn || root->val >= mx) return false;
        return isValidBST(root->left, mn, root->val) && isValidBST(root->right,
            root->val, mx);
    }
};

// 2. Recursion with inorder traversal
class Solution {
public:
    bool isValidBST(TreeNode *root) {
        if (!root) return true;
        vector<int> vals;
        inorder(root, vals);
```

```
        for (int i = 0; i < vals.size() - 1; ++i) {
            if (vals[i] >= vals[i + 1]) return false;
        }
        return true;
    }
    void inorder(TreeNode *root, vector<int> &vals) {
        if (!root) return;
        inorder(root->left, vals);
        vals.push_back(root->val);
        inorder(root->right, vals);
    }
};
```

7.4.3 Binary Search Tree Iterator (M)

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST. Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class BSTIterator {
public:
    BSTIterator(TreeNode *root) {
        pushLeft(root);
    }

    void pushLeft(TreeNode *root) {
        TreeNode *p = root;
        while (p) {
            s.push(p);
            p = p->left;
        }
    }

    /** @return whether we have a next smallest number */
```

```
    bool hasNext() {
        return !s.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *p = s.top();
        s.pop();
        if (p->right)
            pushLeft(p->right);
        return p->val;
    }

private:
    stack<TreeNode*> s;
};

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = BSTIterator(root);
 * while (i.hasNext()) cout << i.next();
 */
```

7.4.4 Recover Binary Search Tree (H)

Two elements of a binary search tree (BST) are swapped by mistake. Recover the tree without changing its structure.

Note: A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

```
// 1. O(n) space complexity
class Solution {
public:
    void recoverTree(TreeNode *root) {
        vector<TreeNode*> list;
        vector<int> vals;
        inorder(root, list, vals);
        sort(vals.begin(), vals.end());
        for (int i = 0; i < list.size(); ++i) {
            list[i]->val = vals[i];
        }
    }
    void inorder(TreeNode *root, vector<TreeNode*> &list, vector<int> &vals) {
        if (!root) return;
```

```
        inorder(root->left, list, vals);
        list.push_back(root);
        vals.push_back(root->val);
        inorder(root->right, list, vals);
    }
};

// 2. O(1) space complexity
class Solution {
public:
    void recoverTree(TreeNode *root) {
        TreeNode *first = NULL, *second = NULL, *parent = NULL;
        TreeNode *cur, *pre;
        cur = root;
        while (cur) {
            if (!cur->left) {
                if (parent && parent->val > cur->val) {
                    if (!first) first = parent;
                    second = cur;
                }
                parent = cur;
                cur = cur->right;
            } else {
                pre = cur->left;
                while (pre->right && pre->right != cur) pre = pre->right;
                if (!pre->right) {
                    pre->right = cur;
                    cur = cur->left;
                } else {
                    pre->right = NULL;
                    if (parent->val > cur->val) {
                        if (!first) first = parent;
                        second = cur;
                    }
                    parent = cur;
                    cur = cur->right;
                }
            }
        }
        if (first && second) swap(first->val, second->val);
    }
};
```

7.4.5 Kth Smallest Element in a BST (M)

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note: You may assume k is always valid, $1 \leq k \leq$BST's total elements.

Follow up: What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Hint:
Try to utilize the property of a BST.
What if you could modify the BST node's structure?
The optimal runtime complexity is O(height of BST).

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        int count = CountNodes(root->left);
        if (k <= count)
            return kthSmallest(root->left, k);
        else if (k > count + 1)
            return kthSmallest(root->right, k-1-count);
        return root->val;
    }

    int CountNodes(TreeNode *node) {
        if (!node) return 0;
        return 1 + CountNodes(node->left) + CountNodes(node->right);
    }
};

// Inorder traversal of BST provides a sorted array of the BST
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        return kthSmallestDFS(root, k);
    }
    int kthSmallestDFS(TreeNode* root, int &k) {
        if (!root) return -1;
        int val = kthSmallestDFS(root->left, k);
        if (k == 0) return val;
        --k;
        if (k == 0) return root->val;
        return kthSmallestDFS(root->right, k);
    }
};
```

7.4.6 Verify Preorder Sequence in Binary Search Tree (M)

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up: Could you do it using only constant space complexity?

```
// e.g. [4, 2, 1, 3, 6, 5, 7]
class Solution {
public:
    bool verifyPreorder(vector<int>& preorder) {
        return helper(preorder, 0, preorder.size() - 1, INT_MIN, INT_MAX);
    }
    bool helper(vector<int> &preorder, int start, int end, int lower, int upper) {
        if (start > end) return true;
        int val = preorder[start], i = 0; // save root value to val
        if (val <= lower || val >= upper) return false;
        for (i = start + 1; i <= end; ++i) {
            if (preorder[i] >= val) break; // break if found the right tree node
        }
        // verify if both the left subtree and the right subtree are valid
        return helper(preorder, start + 1, i - 1, lower, val) && helper(preorder,
            i, end, val, upper);
    }
};
```

7.4.7 Inorder Successor in BST (M)

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return null.

```
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        TreeNode *res = NULL;
        while (root) {
            if (p->val < root->val) {
                res = root; // root could be the in-order successor of p
                root = root->left; // or the successor could exist in the left
                    subtree of root
```

```
            } else { // otherwise, the successor will definitely exist in the
                right subtree of root
                root = root->right;
            }
        }
        return res;
    }
};
```

7.4.8 Largest BST Subtree (M)

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.

Note: A subtree must include all of its descendants.

Here's an example: [10, 5, 15, 1, 8, #, 7]
The Largest BST Subtree in this case is the highlighted one.
The return value is the subtree's size, which is 3.

Hint: You can recursively use algorithm similar to 98. Validate Binary Search Tree at each node of the tree, which will result in O(nlogn) time complexity.

```
class Solution {
public:
    int largestBSTSubtree(TreeNode* root) {
        int res = 0;
        dfs(root, res);
        return res;
    }
    void dfs(TreeNode *root, int &res) {
        if (!root) return;
        int d = countBFS(root, INT_MIN, INT_MAX);
        if (d != -1) {
            res = max(res, d);
            return;
        }
        dfs(root->left, res);
        dfs(root->right, res);
    }
    int countBFS(TreeNode *root, int mn, int mx) {
        if (!root) return 0;
        if (root->val < mn || root->val > mx) return -1;
        int left = countBFS(root->left, mn, root->val);
        if (left == -1) return -1;
```

```
        int right = countBFS(root->right, root->val, mx);
        if (right == -1) return -1;
        return left + right + 1;
    }
};
```

7.4.9 Unique Binary Search Trees (M)

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

```
// Catalan number
// dp[2] = dp[0] * dp[1] + dp[1] * dp[0];
// dp[3] = dp[0] * dp[2] + dp[1] * dp[1] + dp[2] * dp[0];
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n + 1, 0);
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; ++i) {
            for (int j = 0; j < i; ++j) {
                dp[i] += dp[j] * dp[i - j - 1];
            }
        }
        return dp[n];
    }
};
```

7.4.10 Unique Binary Search Trees II (M)

Given an integer n, generate all structurally unique BST's (binary search trees) that store values 1...n.

```
class Solution {
public:
    vector<TreeNode*> generateTrees(int n) {
        if (n == 0) return {};
        return generateTreesDFS(1, n);
    }
    vector<TreeNode *> generateTreesDFS(int start, int end) {
        vector<TreeNode*> subTree;
        if (start > end) {
            subTree.push_back(NULL);
            return subTree;
        }
        for (int k = start; k <= end; k++) {
```

```
            vector<TreeNode*> leftSubTree = generateTreesDFS(start, k - 1);
            vector<TreeNode*> rightSubTree = generateTreesDFS(k + 1, end);
            for (int i = 0; i < leftSubTree.size(); ++i) {
                for (int j = 0; j < rightSubTree.size(); ++j) {
                    TreeNode *node = new TreeNode(k);
                    node->left = leftSubTree[i];
                    node->right = rightSubTree[j];
                    subTree.push_back(node);
                }
            }
        }
        return subTree;
    }
};
```

7.4.11 Convert Sorted Array to Binary Search Tree (M)

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

```
class Solution {
public:
    TreeNode *sortedArrayToBST(vector<int> &nums) {
        return sortedArrayToBST(nums, 0, nums.size()-1);
    }
    TreeNode *sortedArrayToBST(vector<int> &nums, int left, int right) {
        if (left > right) return NULL;
        int mid = left + (right - left) / 2;
        TreeNode *root = new TreeNode(nums[mid]);
        root->left = sortedArrayToBST(nums, left, mid-1);
        root->right = sortedArrayToBST(nums, mid+1, right);
        return root;
    }
};
```

7.4.12 Convert Sorted List to Binary Search Tree (M)

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

```
class Solution {
public:
    TreeNode *sortedListToBST(ListNode *head) {
        if (!head) return NULL;
```

```
        if (!head->next) return new TreeNode (head->val);
        ListNode *left, *mid, *right;
        left = mid = right = head;
        while (right->next && right->next->next) {
            left = mid; // update the tail of left
            mid = mid->next;
            right = right->next->next;
        }
        right = mid->next; // get the right list
        left->next = NULL; // break the left list
        TreeNode *root = new TreeNode(mid->val);
        if (head != mid) root->left = sortedListToBST(head);
        root->right = sortedListToBST(right);
        return root;
    }
};
```

7.4.13 Closest Binary Search Tree Value (E)

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:
Given target value is a floating point.
You are guaranteed to have only one unique value in the BST that is closest to the target.

```
class Solution {
public:
    int closestValue(TreeNode* root, double target) {
        int res = root->val;
        while (root) {
            if (abs(root->val - target) < abs(res - target)) { // find a shorter
                distance
                res = root->val;                              // update value;
            }

            if (target < root->val) {
                root = root->left;
            } else {
                root = root->right;
            }
        }
        return res;
    }
};
```

7.4.14 Closest Binary Search Tree Value II (H)

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

Note:
Given target value is a floating point.
You may assume k is always valid, that is: k ? total nodes.
You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Follow up: Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?

Hint:
1. Consider implement these two helper functions:
i. getPredecessor(N), which returns the next smaller node to N.
ii. getSuccessor(N), which returns the next larger node to N.
2. Try to assume that each node has a parent pointer, it makes the problem much easier.
3. Without parent pointer we just need to keep track of the path from the root to the current node using a stack.
4. You would need two stacks to track the path in finding predecessor and successor node separately.

```
class Solution {
public:
    vector<int> closestKValues(TreeNode* root, double target, int k) {
        vector<int> res, v;
        inorder(root, v);
        int idx = 0;
        double diff = numeric_limits<double>::max();
        for (int i = 0; i < v.size(); ++i) {
            if (diff >= abs(target - v[i])) {
                diff = abs(target - v[i]);
                idx = i;
            }
        }
        int left = idx - 1, right = idx + 1;
        for (int i = 0; i < k; ++i) {
            res.push_back(v[idx]);
            if (left >= 0 && right < v.size()) {
                if (abs(v[left] - target) > abs(v[right] - target)) {
                    idx = right;
                    ++right;
                } else {
                    idx = left;
```

```
                --left;
            }
        } else if (left >= 0) {
            idx = left;
            --left;
        } else if (right < v.size()) {
            idx = right;
            ++right;
        }
    }
    return res;
}
void inorder(TreeNode *root, vector<int> &v) {
    if (!root) return;
    inorder(root->left, v);
    v.push_back(root->val);
    inorder(root->right, v);
}
};
```

Chapter 8

Sorting

8.1 Sort Colors (M)

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?

```
// 1. Count and Sort
class Solution {
public:
    void sortColors(vector<int> &nums) {
        int count[3] = {0}, idx = 0;
        // Count the number of each color
        for (int i = 0; i < nums.size(); ++i) ++count[nums[i]];
        // Sort the array by assigning correct number of colors based on counts
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < count[i]; ++j) {
                nums[idx++] = i;
            }
        }
    }
};

// 2. Two pointers
class Solution {
```

```
public:
    void sortColors(vector<int> &nums) {
        int red = 0, blue = nums.size()-1, i = 0;
        while (i < blue + 1) {
            if (nums[i] == 0) swap(nums[i++], nums[red++]);
            // nums[i] still need to be checked in the next loop after swap
            else if (nums[i] == 2) swap(nums[i], nums[blue--]);
            else ++i;
        }
    }
};
```

8.2 Wiggle Sort (M)

Given an unsorted array nums, reorder it in-place such that $nums[0] <= nums[1] >= nums[2] <= nums[3]....$

For example, given nums = [3, 5, 2, 1, 6, 4], one possible answer is [1, 6, 2, 5, 3, 4].

```
// Wiggle Sort O(nlogn)
class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        if (nums.size() <= 2) return;
        for (int i = 2; i < num.size(); i += 2) {
            swap(nums[i], nums[i-1]);
        }
    }
};


// Wiggle Sort O(n)
class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        if (nums.size() <= 1) return;
        for (int i = 1; i < nums.size(); ++i) {
            if ( (i % 2 == 1 && nums[i] < nums[i-1]) ||
                 (i % 2 == 0 && nums[i] > nums[i-1]) ) {
                    swap(nums[i], nums[i-1]);
                 }
        }
    }
};
```

8.3 Wiggle Sort II (M)

Given an unsorted array nums, reorder it such that $nums[0] < nums[1] > nums[2] < nums[3]$....

Example:
(1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is [1, 4, 1, 5, 1, 6].
(2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is [2, 3, 1, 3, 1, 2].

Note: You may assume all input has valid answer.

Follow Up: Can you do it in O(n) time and/or in-place with O(1) extra space?

```
// Wiggle Sort II Time: O(nlogn) Space: O(n)
class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        vector<int> tmp = nums;
        int n = nums.size(), k = (n+1)/2, j = n;
        sort(tmp.begin(), tmp.end());
        for (int i = 0; i < n; ++i) {
            if (i % 2 == 0) {
                nums[i] = tmp[--k];
            } else {
                nums[i] = tmp[--j];
            }
        }
    }
};
```

8.4 Wiggle Subsequence (M)

A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.

For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because its last difference

is zero.

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:

Input: [1,7,4,9,2,5]
Output: 6
The entire sequence is a wiggle sequence.

Input: [1,17,5,10,13,15,10,5,16,8]
Output: 7
There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8].

Input: [1,2,3,4,5,6,7,8,9]
Output: 2

Follow up: Can you do it in O(n) time?

```cpp
// O(n^2)
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        if (nums.empty()) return 0;
        int n = nums.size();
        vector<int> p(n, 1);
        vector<int> q(n, 1);

        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[i] > nums[j]) {
                    p[i] = max(p[i], q[j] + 1);
                } else if (nums[i] < nums[j]) {
                    q[i] = max(q[i], p[j] + 1);
                } else {
                    continue;
                }
            }
        }

        return max(p.back(), q.back());
    }
};
```

```
// O(n)
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int p = 1, q = 1, n = nums.size();

        for (int i = 1; i < n; ++i) {
            if (nums[i] > nums[i - 1]){
                p = q + 1;
            } else if (nums[i] < nums[i - 1]) {
                q = p + 1;
            }
        }

        return min(n, max(p, q));
    }
};
```

8.5 Sort List (M)

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

```
// Merge Sort
class Solution {
public:
    ListNode *sortList(ListNode *head) {
        if (!head || !head->next) return head;
        ListNode *p1 = head, *p2 = head;
        while (p2->next && p2->next->next) {
            p1 = p1->next;
            p2 = p2->next->next;
        }
        p2 = sortList(p1->next);
        p1->next = NULL;
        p1 = sortList(head);
        return mergeList(p1, p2);
    }
    ListNode *mergeList(ListNode *p1, ListNode *p2) {
        ListNode *new_head = new ListNode(0);
        ListNode *tmp = new_head;
        while (p1 && p2) {
            if (p1->val < p2->val) {
                tmp->next = p1;
                p1 = p1->next;
```

```
            } else {
                tmp->next = p2;
                p2 = p2->next;
            }
            tmp = tmp->next;
        }
        if (p1) tmp->next = p1;
        if (p2) tmp->next = p2;
        return new_head->next;
    }
};
```

8.6 Insertion Sort List (M)

Sort a linked list using insertion sort.

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *new_head = new ListNode(0);
        ListNode *cur, *next;
        while (head) {
            cur = new_head; // reset cur to the beginning for each iteration
            next = head->next; // next point to the next position of head
            // search the correct position to insert head
            while (cur->next && cur->next->val <= head->val) {
                cur = cur->next;
            }
            head->next = cur->next; // add head after cur
            cur->next = head; // update cur->next
            head = next; // update head
        }
        return new_head->next;
    }
};
```

Chapter 9

Search

9.1 Binary Search

9.1.1 First Bad Version (E)

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad. You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

```
// Forward declaration of isBadVersion API.
bool isBadVersion(int version);

class Solution {
public:
    int firstBadVersion(int n) {
        int start = 1, end = n, mid;
        while (start < end) {
            mid = start + (end - start) / 2;
            if (!isBadVersion(mid))
                start = mid + 1;
            else
                end = mid;        // the current mid could be the first bad version
        }
        return start;
    }
};
```

9.1.2 Search Insert Position (M)

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You may assume no duplicates in the array.

Here are few examples.
$[1,3,5,6], 5 -> 2$
$[1,3,5,6], 2 -> 1$
$[1,3,5,6], 7 -> 4$
$[1,3,5,6], 0 -> 0$

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (target < nums[mid]) {
                right = mid - 1;
            } else if (target > nums[mid]) {
                left = mid + 1;
            } else {
                return mid;
            }
        }
        return left;
    }
};
```

9.1.3 Search a 2D Matrix (M)

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 m = matrix.size(), n = matrix[0].size();
```

```
        int left = 0, right = m * n - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int val = matrix[mid/n][mid%n];// convert 1D index to 2D index
            if (target == val)
                return true;
            else if (target > val)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return false;
    }
};
```

9.1.4 Search a 2D Matrix II (M)

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 in ascending from left to right.
Integers in each column are sorted in ascending from top to bottom.

For example, Consider the following matrix: [[1, 4, 7, 11, 15], [2, 5, 8, 12, 19], [3, 6, 9, 16, 22], [10, 13, 14, 17, 24], [18, 21, 23, 26, 30]]
Given target = 5, return true. Given target = 20, return false.

```
// 1. Binary search for each row: O(MlogN)
class Solution {
public:
    bool searchMatrix(vector<vector<int>> &matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        bool res = false;
        for (int i = 0; i < m; ++i) {
            if (matrix[i][0] <= target && target <= matrix[i][n-1]) {
                res = binarySearch(matrix, target, i, n);
                if (res == true) break;
            }
        }
        return res;
    }
    bool binarySearch(vector<vector<int>> &matrix, int target, int row, int
        length) {
        int left = 0, right = length-1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
```

```
            if (target == matrix[row][mid]) {
                return true;
            } else if (target > matrix[row][mid]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return false;
    }
};

// 2. O(M+N) solution
// Starting from a corner of matrix,
// if one direction is ascending and another is decending,
// then this method works!
// e.g. for this case, starting from the bottom left corner or upper right corner
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        int i = 0, j = n - 1;
        // starting from the upper right corner
        while(i < m && j >= 0) {
            if (target == matrix[i][j])
                return true;
            else if (target < matrix[i][j])
                --j;
            else
                ++i;
        }
        return false;
    }
};
```

9.1.5 Kth Smallest Element in a Sorted Matrix (M)

Given a n x n matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.
Note that it is the kth smallest element in the sorted order, not the kth distinct element.

Example:
matrix = [[1, 5, 9], [10, 11, 13], [12, 13, 15]],
k = 8, return 13.

Note: You may assume k is always valid, $1 \leq k \leq n^2$.

```cpp
// 1. Binary search
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int left = matrix[0][0], right = matrix.back().back();
        while (left < right) {
            int mid = left + (right - left) / 2, cnt = 0;
            // upper_bound(): Returns an iterator pointing to the first element
            // in the range [first,last) which compares greater than val.
            for (int i = 0; i < matrix.size(); ++i) {
                // get the position of upper_bound compared to mid
                cnt += upper_bound(matrix[i].begin(), matrix[i].end(), mid) -
                    matrix[i].begin();
            }
            if (cnt < k) left = mid + 1;
            else right = mid;
        }
        return left;
    }
};

// 2. Heap
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        priority_queue<int, vector<int>> q;
        for (int i = 0; i < matrix.size(); ++i) {
            for (int j = 0; j < matrix[i].size(); ++j) {
                q.emplace(matrix[i][j]);
                if (q.size() > k) q.pop();
            }
        }
        return q.top();
    }
};
```

9.1.6 Guess Number Higher or Lower (E)

We are playing the Guess Game. The game is as follows: I pick a number from 1 to n. You have to guess which number I picked. Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):
-1 : My number is lower

1 : My number is higher
0 : Congrats! You got it!

Example:
n = 10, I pick 6.
Return 6.

```
// Forward declaration of guess API.
// @param num, your guess
// @return -1 if my number is lower, 1 if my number is higher, otherwise return 0
int guess(int num);

class Solution {
public:
    int guessNumber(int n) {
        int start = 1, end = n;

        while (start <= end) {
            int mid = start + (end - start) / 2;
            int res = guess(mid);

            if (res == 0)
                return mid;
            else if (res == 1)
                start = mid + 1;
            else
                end = mid - 1;
        }

        return start;
    }
};
```

9.1.7 Guess Number Higher or Lower II (M)

We are playing the Guess Game. The game is as follows: I pick a number from 1 to n. You have to guess which number I picked. Every time you guess wrong, I'll tell you whether the number I picked is higher or lower. However, when you guess a particular number x, and you guess wrong, you pay $x. You win the game when you guess the number I picked.

Example:
n = 10, I pick 8.
First round: You guess 5, I tell you that it's higher. You pay $5.
Second round: You guess 7, I tell you that it's higher. You pay $7.
Third round: You guess 9, I tell you that it's lower. You pay $9.

Game over. 8 is the number I picked.
You end up paying $5 + $7 + $9 = $21.

Given a particular n $\geq$ 1, find out how much money you need to have to guarantee a win.

```
class Solution {
public:
    int getMoneyAmount(int n) {
        vector<vector<int>> dp(n+1, vector<int>(n+1, 0));
        return solver(dp, 1, n);
    }

    int solver(vector<vector<int>> &dp, int L, int R) {
        if (L >= R) return 0;
        if (dp[L][R]) return dp[L][R];

        dp[L][R] = INT_MAX;

        // f(x) = x + max(solver(L,x-1),solver(x+1,n))
        // get the minimum f(x) for x = 1~n
        for (int i = L; i <= R; ++i) {
            dp[L][R] = min(dp[L][R], i + max(solver(dp, L, i-1), solver(dp, i+1,
                R)));
        }

        return dp[L][R];
    }
};
```

9.1.8 Find Minimum in Rotated Sorted Array (M)

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.

```
class Solution {
public:
    int findMin(vector<int> &nums) {
        int left = 0, right = nums.size()-1;
        // If nums[left] > nums[right], then the array/subarray must be rotated
        // Otherwise, the array/subarray is not rotated and nums[left] is the
            minimum
        while (left < right && nums[left] > nums[right]) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) { //min must exist in right part
```

```
            left = mid + 1;
        } else { //otherwise min is in the left part
            right = mid;
        }
    }
    return nums[left];
  }
};
```

9.1.9 Find Minimum in Rotated Sorted Array II (H)

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. The array may contain duplicates.

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int left = 0, right = nums.size()-1;
        while (left < right && nums[left] >= nums[right]) { //consider deplicates
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right])
                left = mid + 1;
            else if (nums[mid] < nums[right])
                right = mid;
            else //if duplicates exist, skip the leftmost element and procees to
                the next
                left = left + 1;
        }
        return nums[left];
    }
};
```

9.1.10 Search in Rotated Sorted Array (H)

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.

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size()-1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
```

```
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < nums[right]) { //right part is sorted
                if (nums[mid] < target && target <= nums[right]) { //target exists
                    in the right part
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            } else { //left part is sorted
                if (nums[left] <= target && target < nums[mid]) { //target exists
                    in the left part
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
        }
        return -1;
    }
};
```

9.1.11 Search in Rotated Sorted Array II (M)

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.

```
class Solution {
public:
    bool search(vector<int> &nums, int target) {
        int left = 0, right = nums.size()-1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                return true;
            } else if (nums[mid] < nums[right]) {
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            } else if (nums[mid] > nums[right]) {
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
```

```
                left = mid + 1;
            }
        } else { //skip duplicates by leftshift the right index
            --right;
        }
    }
    return false;
  }
};
```

9.2 Deep-first Search

9.2.1 Walls and Gates (M)

You are given a m x n 2D grid initialized with these three possible values.
-1 - A wall or an obstacle.
0 - A gate.
INF - Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

For example, given the 2D grid:
INF -1 0 INF
INF INF INF -1
INF -1 INF -1
0 -1 INF INF

After running your function, the 2D grid should be:
3 -1 0 1
2 2 1 -1
1 -1 2 -1
0 -1 3 4

```
class Solution {
public:
    void wallsAndGates(vector<vector<int>>& rooms) {
        for (int i = 0; i < rooms.size(); ++i) {
            for (int j = 0; j < rooms[i].size(); ++j) {
                if (rooms[i][j] == 0) {
                    dfs(rooms, i, j, 0);
                }
            }
```

```
        }
    }
    void dfs(vector<vector<int>> &rooms, int i, int j, int val) {
        if (i < 0 || i >= rooms.size() || j < 0 || j >= rooms[i].size() ||
            rooms[i][j] < val) return;
        rooms[i][j] = val;
        dfs(rooms, i + 1, j, val + 1);
        dfs(rooms, i - 1, j, val + 1);
        dfs(rooms, i, j + 1, val + 1);
        dfs(rooms, i, j - 1, val + 1);
    }
};
```

9.2.2 Combinations (M)

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]]

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> res;
        vector<int> out;
        if (k > n) return res;
        combineDFS(res, out, 1, n, k);
        return res;
    }
    void combineDFS(vector<vector<int>> &res, vector<int> &out, int start, int
        end, int k) {
        if (k == 0) res.push_back(out);
        for (int i = start; i <= end; ++i) {
            out.push_back(i);
            combineDFS(res, out, i+1, end, k-1);
            out.pop_back();
        }
    }
};
```

9.2.3 Combination Sum (M)

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.
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) {
        vector<vector<int>> res;
        vector<int> out;
        sort(candidates.begin(), candidates.end());
        combinationSumDFS(candidates, target, out, res, 0);
        return res;
    }
    void combinationSumDFS(vector<int>& candidates, int target, vector<int>& out,
        vector<vector<int>> &res, int index) {
        if (target < 0) {
            return;
        } else if (target == 0) {
            res.push_back(out);
        } else {
            for (int i = index; i < candidates.size(); ++i) {
                out.push_back(candidates[i]);
                combinationSumDFS(candidates, target-candidates[i], out, res, i);
                out.pop_back();
            }
        }
    }
};
```

9.2.4 Combination Sum II (M)

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.
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]]

```
class Solution {
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        vector<int> out;
        sort(candidates.begin(), candidates.end());
        combinationSum2DFS(candidates, target, out, res, 0);
        return res;
    }
    void combinationSum2DFS(vector<int>& candidates, int target, vector<int>&
        out, vector<vector<int>> &res, int index) {
        if (target < 0) {
            return;
        } else if (target == 0) {
            res.push_back(out);
        } else {
            for (int i = index; i < candidates.size(); ++i) {
                if (i == index || candidates[i] != candidates[i-1]) { //skip
                    duplicates
                    out.push_back(candidates[i]);
                    combinationSum2DFS(candidates, target-candidates[i], out, res,
                        i+1);//update i to i+1
                    out.pop_back();
                }
            }
        }
    }
};
```

9.2.5 Combination Sum III (M)

Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Example 1:
Input: k = 3, n = 7
Output: [[1,2,4]]

Example 2:
Input: k = 3, n = 9
Output:[[1,2,6], [1,3,5], [2,3,4]]

```
class Solution {
public:
```

```
    vector<vector<int>> combinationSum3(int k, int n) {
        vector<vector<int>> res;
        vector<int> out;
        combinationSum3DFS(k, n, out, res, 1);
        return res;
    }
    void combinationSum3DFS(int k, int n, vector<int>& out, vector<vector<int>>
        &res, int index) {
        if (n < 0) {
            return;
        } else if (k == 0 && n == 0) {
            res.push_back(out);
        } else {
            for (int i = index; i <= 9; ++i) {
                out.push_back(i);
                combinationSum3DFS(k-1, n-i, out, res, i+1);
                out.pop_back();
            }
        }
    }
};
```

9.2.6 Factor Combinations (M)

Numbers can be regarded as product of its factors. For example, 8 = 2 x 2 x 2 = 2 x 4.

Write a function that takes an integer n and return all possible combinations of its factors.

Note:
Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is [2, 6], not [6, 2].
You may assume that n is always positive.
Factors should be greater than 1 and less than n.

Examples:
input: 1 output: []

input: 37 output: []

input: 12 output: [[2, 6], [2, 2, 3], [3, 4]]

input: 32 output: [[2, 16], [2, 2, 8], [2, 2, 2, 4], [2, 2, 2, 2, 2], [2, 4, 4], [4, 8]]

```
class Solution {
public:
```

```
    vector<vector<int>> getFactors(int n) {
        vector<vector<int>> res;
        dfs(n, 2, {}, res);
        return res;
    }
    void dfs(int n, int start, vector<int> out, vector<vector<int>> &res) {
        if (n == 1) {
            if (out.size() > 1) res.push_back(out);
        } else {
            for (int i = start; i <= n; ++i) {
                if (n % i == 0) {
                    out.push_back(i);
                    dfs(n / i, i, out, res);
                    out.pop_back();
                }
            }
        }
    }
};
```

9.2.7 Permutations (M)

Given a collection of distinct 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], [3,2,1]]

```
// 1. Use visited vector
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        vector<vector<int>> res;
        vector<int> out;
        vector<int> visited(num.size(), 0); // save visited states
        permuteDFS(num, 0, visited, out, res);
        return res;
    }
    void permuteDFS(vector<int> &num, int level, vector<int> &visited,
        vector<int> &out, vector<vector<int>> &res) {
        if (level == num.size()) res.push_back(out);
        else {
            for (int i = 0; i < num.size(); ++i) {
                if (visited[i] == 0) {
                    visited[i] = 1;
                    out.push_back(num[i]);
```

```
                permuteDFS(num, level + 1, visited, out, res);
                out.pop_back();
                visited[i] = 0;
            }
        }
    }
};

// 2. Use swap function
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> res;
        permuteDFS(nums, 0, nums.size()-1, res);
        return res;
    }
    void permuteDFS(vector<int>& nums, int start, int end, vector<vector<int>>
        &res) {
        if (start > end) {
            res.push_back(nums);
            return;
        }
        for (int i = start; i <= end; ++i) {
            swap(nums[start], nums[i]);
            permuteDFS(nums, start+1, end, res);
            swap(nums[start], nums[i]);
        }
    }
};
```

9.2.8 Permutations II (M)

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], [2,1,1]]

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int> &num) {
        vector<vector<int>> res;
        vector<int> out;
        vector<int> visited(num.size(), 0);
        sort(num.begin(), num.end());
```

```
        permuteUniqueDFS(num, 0, visited, out, res);
        return res;
    }
    void permuteUniqueDFS(vector<int> &num, int level, vector<int> &visited,
        vector<int> &out, vector<vector<int>> &res) {
        if (level >= num.size()) res.push_back(out);
        else {
            for (int i = 0; i < num.size(); ++i) {
                if (visited[i] == 0) {
                    // skip duplicates
                    if (i > 0 && num[i] == num[i-1] && visited[i-1] == 0) continue;
                    visited[i] = 1;
                    out.push_back(num[i]);
                    permuteUniqueDFS(num, level + 1, visited, out, res);
                    out.pop_back();
                    visited[i] = 0;
                }
            }
        }
    }
};

class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> res;
        permuteUniqueDFS(nums, 0, nums.size()-1, res);
        return res;
    }
    // Do not use reference of nums and do not swap back after recursion
    void permuteUniqueDFS(vector<int> nums, int start, int end,
        vector<vector<int>> &res) {
        if (start > end) {
            res.push_back(nums);
            return;
        }
        for (int i = start; i <= end; ++i) {
            if (i != start && nums[start] == nums[i]) continue;
            swap(nums[start], nums[i]);
            permuteUniqueDFS(nums, start+1, end, res);
        }
    }
};
```

9.2.9 Next Permutation (M)

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers. If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order). The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.
1,2,3 $->$ 1,3,2
3,2,1 $->$ 1,2,3
1,1,5 $->$ 1,5,1

```
// 1. Find the first num that is smaller than 7:
//     1   2#  7   4   3   1
// 2. Find the first num that is larger than 2:
//     1   2   7   4   3#  1
// 3. Swap 2 and 3:
//     1   3#  7   4   2#  1
// 4. Reverse the left numbers after 3:
//     1   3   1#  2#  4#  7#
class Solution {
public:
    void nextPermutation(vector<int> &num) {
        int i, j, n = num.size();
        for (i = n - 2; i >= 0; --i) {
            if (num[i + 1] > num[i]) { // step 1
                for (j = n - 1; j >= i; --j) {
                    if (num[j] > num[i]) break; // step 2
                }
                swap(num[i], num[j]); // step 3
                reverse(num.begin() + i + 1, num.end()); // step 4
                return;
            }
        }
        reverse(num.begin(), num.end()); // reverse nums if no next permutation
    }
};
```

9.2.10 Permutation Sequence (M)

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 (i.e., for n = 3): "123"
"132"
"213"

”231”
”312”
”321”
Given n and k, return the kth permutation sequence.
Note: Given n will be between 1 and 9 inclusive.

```
class Solution {
public:
    string getPermutation(int n, int k) {
        string res;
        string num = "123456789";
        vector<int> f(n, 1);
        for (int i = 1; i < n; ++i) {
            f[i] = f[i - 1] * i; // compute 1!, 2!, ...
        }
        --k; // align index
        for (int i = n; i >= 1; --i) {
            int j = k / f[i - 1];
            k %= f[i - 1];
            res.push_back(num[j]);
            num.erase(j, 1);
        }
        return res;
    }
};
```

9.3 Breadth-first Search

Chapter 10

Dynamic Programming

10.1 Climbing Stairs (E)

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> steps(n,0);
        steps[0] = 1;
        steps[1] = 2;
        // the number of distinct ways to reach level n is the sum of number of
           distinct ways to reach level n-1 and n-2.
        for(int i = 2; i < n; i++)
            steps[i] = steps[i-2] + steps[i-1];
        return steps[n-1];
    }
};
```

10.2 Combination Sum IV (M)

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

Example: nums = [1, 2, 3], target = 4
The possible combination ways are:
(1, 1, 1, 1) (1, 1, 2) (1, 2, 1) (1, 3) (2, 1, 1) (2, 2) (3, 1)

Note that different sequences are counted as different combinations. Therefore the output is 7.

Follow up:
What if negative numbers are allowed in the given array?
How does it change the problem?
What limitation we need to add to the question to allow negative numbers?

```
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        vector<int> dp(target+1);
        dp[0] = 1;
        for (int i = 1; i <= target; ++i) {
            for (auto a : nums) {
                if (i >= a) dp[i] += dp[i-a];
            }
        }
        return dp.back();
    }
};
```

10.3 Perfect Squares (M)

Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.

For example, given n = 12, return 3 because $12 = 4 + 4 + 4$; given n = 13, return 2 because $13 = 4 + 9$.

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;

        // if x = a + b * b, the least number of perfect square numbers which sum
            to x is dp[x], then
        // case1: dp[x] = dp[a] + 1, because b * b is a perfect square number
        // case2: dp[x] = dp[a + b*b], because a + b * b is a perfect square
            number
        // dp[x] = min(case1, case2)
        for (int i = 0; i <= n; ++i) {
            for (int j = 1; i + j * j <= n; ++j) {
```

```
                dp[i + j * j] = min(dp[i + j * j], dp[i] + 1);
            }
        }
        return dp[n];
    }
};
```

10.4 Best Time to Buy and Sell Stock (E)

Say you have an array for which the i-th 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) {
        if (prices.empty() || prices.size() < 2)
            return 0;

        int profit = 0;
        int low = prices[0];

        for (int i = 1; i < prices.size(); ++i) {
            profit = max(profit, prices[i] - low);
            low = min(low, prices[i]);
        }

        return profit;
    }
};
```

10.5 Best Time to Buy and Sell Stock II (M)

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).

```
class Solution {
public:
```

```
    int maxProfit(vector<int>& prices) {
        if (prices.empty() || prices.size() < 2)
            return 0;

        int profit = 0;
        int diff;

        for (int i = 1; i < prices.size(); ++i) {
            diff = prices[i] - prices[i-1];
            if (diff > 0)
                profit += diff;
        }

        return profit;
    }
};
```

10.6 Best Time to Buy and Sell Stock with Cooldown (M)

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) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {

        int buy = INT_MIN, sell = 0, rest = INT_MIN, cooldown = 0;

        for (int i = 0; i < prices.size(); ++i) {
            rest = max(rest, buy);
            buy = cooldown - prices[i];
            cooldown = max(sell, cooldown);
            sell = rest + prices[i];
        }

        return max(cooldown, sell);
```

```
    }
};
```

10.7 Best Time to Buy and Sell Stock III (H)

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. However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty() || prices.size() < 2)
            return 0;

        vector<int> profit(prices.size());

        // compute the forward max profit and save it
        int buy = prices[0];
        profit[0] = 0;
        for (int i = 1; i < prices.size(); i++) {
            profit[i] = max(profit[i - 1], prices[i] - buy);
            buy = min(buy, prices[i]);
        }

        // The final max profit is the sum of max profit before day i (profit[i])
            and after day i (sell - prices[i])
        int sell = prices[prices.size() - 1];
        int best = 0;
        for (int i = prices.size() - 2; i >= 0; i--) {
            best = max(best, sell - prices[i] + profit[i]);
            sell = max(sell, prices[i]);
        }

        return best;
    }
};
```

10.8 Best Time to Buy and Sell Stock IV (H)

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 k transactions. However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

```
class Solution {
public:
    int maxProfit(int k, vector<int> &prices) {
        if (prices.empty() || prices.size() < 2)
            return 0;
        if (k >= prices.size())
            return solveMaxProfit(prices);

        int global[k + 1] = {0};
        int local[k + 1] = {0};
        for (int i = 0; i < prices.size() - 1; ++i) {
            int diff = prices[i + 1] - prices[i];
            for (int j = k; j >= 1; --j) {
                local[j] = max(global[j - 1] + max(diff, 0), local[j] + diff);
                global[j] = max(global[j], local[j]);
            }
        }
        return global[k];
    }

    int solveMaxProfit(vector<int> &prices) {
        int profit = 0;
        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] > prices[i - 1]) {
                profit += prices[i] - prices[i - 1];
            }
        }
        return profit;
    }
};
```

10.9 House Robber (E)

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.

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int cur_rob = 0, prev_rob = 0, sum = 0;

        for (int i = 0; i < nums.size(); ++i) {
            cur_rob = prev_rob + nums[i];
            prev_rob = sum;
            sum = max(cur_rob, prev_rob);
        }

        return sum;
    }
};
```

10.10 House Robber II (M)

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.

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if (n == 0) return 0;
        if (n == 1) return nums[0];
        return max(rob(nums, 0, n-2), rob(nums, 1, n-1)); // can not rob nums[0]
            and nums[n-1] together
    }

    int rob(vector<int> &nums, int start, int end) {
        int cur_rob = 0, prev_rob = 0, sum = 0;

        for (int i = start; i <= end; ++i) {
```

```
            cur_rob = prev_rob + nums[i];
            prev_rob = sum;
            sum = max(cur_rob, prev_rob);
        }

        return sum;
    }
};
```

10.11 House Robber III (M)

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

```
class Solution {
public:
    int rob(TreeNode *root) {
        vector<int> res = robber(root);
        return max(res[0], res[1]);
    }

    vector<int> robber(TreeNode *root) {
        vector<int> res(2,0);
        if (!root) return res;

        vector<int> left = robber(root->left);
        vector<int> right = robber(root->right);

        res[0] = max(left[0], left[1]) + max(right[0], right[1]); // if root is
            not robbed
        res[1] = root->val + left[0] + right[0];              // if root is robbed

        return res;
    }
};
```

10.12 Paint Fence (E)

There is a fence with n posts, each post can be painted with one of the k colors. You have to paint all the posts such that no more than two adjacent fence posts have the same color. Return the total number of ways you can paint the fence.

Note: n and k are non-negative integers.

```
class Solution {
public:
    int numWays(int n, int k) {
        if (n == 0) return 0;
        if (n == 1) return k;
        vector<int> dp(n);

        dp[0] = k;                          // n = 1, k ways to paint
        dp[1] = k * (k - 1) + k;            // n = 2, diff color: k*(k-1) ways + same
            color: k ways

        /** 1. If the color of the current post i is different from the color of
            the last post i-1,
         *     then there are dp[i] = dp[i - 1] * (k - 1) ways to paint the
             current post i
         *  2. If the color of the current post i is same as the color of the
             last post i-1,
         *     then the color of the post i and i-1 must be different from the
             color of the second last post i-2
         *     so there are dp[i] = dp[i - 2] * (k - 1) * 1 ways to paint the
             current post i
         *  3. The total num of ways is a combination of case 1 and 2
         */
        for (int i = 2; i < n; i++) {
            dp[i] = dp[i - 1] * (k - 1) + dp[i - 2] * (k - 1);
        }

        return dp[n - 1];
    }
};
```

10.13 Paint House (M)

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x 3 cost matrix. For example, costs[0][0] is the cost of painting house 0 with color red; costs[1][2] is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note: All costs are positive integers.

```
class Solution {
public:
    int minCost(vector<vector<int>> &cost) {
        if (cost.empty() || cost[0].empty()) return 0;
        vector<vector<int>> dp = costs;
        for (int i = 1; i < dp.size(); ++i) {
            // dp[i][0] += min(dp[i - 1][1], dp[i - 1][2]);
            // dp[i][1] += min(dp[i - 1][0], dp[i - 1][2]);
            // dp[i][2] += min(dp[i - 1][0], dp[i - 1][1]);
            for (int j = 0; j < 3; ++j) {
                dp[i][j] += min(dp[i-1][(j+1)%3], dp[i-1][(j+2)%3]);
            }
        }
        return min(min(dp.back()[0], dp.back()[1]), dp.back()[2]);
    }
};
```

10.14 Paint House II (H)

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x k cost matrix. For example, costs[0][0] is the cost of painting house 0 with color 0; costs[1][2]is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Note: All costs are positive integers.

Follow up: Could you solve it in O(nk) runtime?

```
class Solution {
public:
    int minCostII(vector<vector<int>>& costs) {
        if(costs.empty() || costs[0].empty()) return 0;
        int n = costs.size(), k = costs[0].size(), res = INT_MAX;
        vector<vector<int>> dp = costs;
```

```
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < k; ++j) {
                int tmp = INT_MAX;
                // find the local min cost of using other color to paint the last
                    house
                for (int d = 1; d < k; ++d) {
                    tmp = min(tmp, dp[i-1][(j+d)%k]);
                }
                dp[i][j] += tmp;
                // find the global min cost of painting all houses
                if (i == n-1) {
                    res = min(res, dp[i][j]);
                }
            }
        }
        return res;
    }
};

class Solution {
public:
    int minCostII(vector<vector<int>>& costs) {
        if (costs.empty() || costs[0].empty()) return 0;
        vector<vector<int>> dp = costs;
        int min1 = -1, min2 = -1;
        for (int i = 0; i < dp.size(); ++i) {
            int last1 = min1, last2 = min2;
            min1 = -1; min2 = -1;
            for (int j = 0; j < dp[i].size(); ++j) {
                if (j != last1) {
                    dp[i][j] += last1 < 0 ? 0 : dp[i - 1][last1];
                } else {
                    dp[i][j] += last2 < 0 ? 0 : dp[i - 1][last2];
                }
                if (min1 < 0 || dp[i][j] < dp[i][min1]) {
                    min2 = min1; min1 = j;
                } else if (min2 < 0 || dp[i][j] < dp[i][min2]) {
                    min2 = j;
                }
            }
        }
        return dp.back()[min1];
    }
};
```

10.15 Pascal's Triangle (E)

Given numRows, generate the first numRows of Pascal's triangle.

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> res;
        if (numRows == 0) return res;

        res.push_back(vector<int> (1,1));          // first row

        for (int i = 2; i <= numRows; ++i) {
            vector<int> cur(i,1);                  // generate next row with all
                1s
            for (int j = 1; j < i-1; ++j)          // update elements from the
                2nd to the (i-1)-th
                cur[j] = res[i-2][j-1] + res[i-2][j];
            res.push_back(cur);
        }

        return res;
    }
};
```

10.16 Pascal's Triangle II (E)

Given an index k, return the k-th row of the Pascal's triangle.

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> res;

        for (int i = 0; i <= rowIndex; ++i) {
            for (int j = i - 1; j > 0; --j) {
                res[j] = res[j-1] + res[j];  // scrolling array
            }
            res.push_back(1);
        }

        return res;
    }
};
```

10.17 Range Sum Query - Immutable (E)

Given an integer array nums, find the sum of the elements between indices i and j ($i \leq j$), inclusive.

Example:
Given nums = [-2, 0, 3, -5, 2, -1]
sumRange(0, 2) = 1
sumRange(2, 5) = -1
sumRange(0, 5) = -3

Note:
You may assume that the array does not change.
There are many calls to sumRange function.

```
// Your NumArray object will be instantiated and called as such:
// NumArray numArray(nums);
// numArray.sumRange(0, 1);
// numArray.sumRange(1, 2);

class NumArray {
public:
    vector<int> sums = {0};          // save an initial 0 into sums

    NumArray(vector<int> &nums) {    // class constructor
        int sum = 0;
        for (int i = 0; i < nums.size(); ++i) {
            sum += nums[i];
            sums.push_back(sum);     // sums[] contains nums.size() + 1 elements
        }
    }

    int sumRange(int i, int j) {
        return sums[j+1] - sums[i];  // get the correct sum by considering the
            offset in sums[]
    }
};
```

10.18 Range Sum Query - Mutable (M)

Given an integer array nums, find the sum of the elements between indices i and j ($i \leq j$), inclusive. The update(i, val) function modifies nums by updating the element at index i to val.

Example:
Given nums = [1, 3, 5] sumRange(0, 2) = 9 update(1, 2) sumRange(0, 2) = 8
Note:
The array is only modifiable by the update function.
You may assume the number of calls to update and sumRange function is distributed evenly.

```
struct SegmentTreeNode {
    int start, end, sum;
    SegmentTreeNode* left;
    SegmentTreeNode* right;
    SegmentTreeNode(int a, int
        b):start(a),end(b),sum(0),left(nullptr),right(nullptr){}
};
class NumArray {
    SegmentTreeNode* root;
public:
    NumArray(vector<int> &nums) {
        int n = nums.size();
        root = buildTree(nums,0,n-1);
    }

    void update(int i, int val) {
        modifyTree(i,val,root);
    }

    int sumRange(int i, int j) {
        return queryTree(i, j, root);
    }
    SegmentTreeNode* buildTree(vector<int> &nums, int start, int end) {
        if(start > end) return nullptr;
        SegmentTreeNode* root = new SegmentTreeNode(start,end);
        if(start == end) {
            root->sum = nums[start];
            return root;
        }
        int mid = start + (end - start) / 2;
        root->left = buildTree(nums,start,mid);
        root->right = buildTree(nums,mid+1,end);
        root->sum = root->left->sum + root->right->sum;
        return root;
    }
    int modifyTree(int i, int val, SegmentTreeNode* root) {
        if(root == nullptr) return 0;
        int diff;
        if(root->start == i && root->end == i) {
            diff = val - root->sum;
```

```
            root->sum = val;
            return diff;
        }
        int mid = (root->start + root->end) / 2;
        if(i > mid) {
            diff = modifyTree(i,val,root->right);
        } else {
            diff = modifyTree(i,val,root->left);
        }
        root->sum = root->sum + diff;
        return diff;
    }
    int queryTree(int i, int j, SegmentTreeNode* root) {
        if(root == nullptr) return 0;
        if(root->start == i && root->end == j) return root->sum;
        int mid = (root->start + root->end) / 2;
        if(i > mid) return queryTree(i,j,root->right);
        if(j <= mid) return queryTree(i,j,root->left);
        return queryTree(i,mid,root->left) + queryTree(mid+1,j,root->right);
    }
};


// Your NumArray object will be instantiated and called as such:
// NumArray numArray(nums);
// numArray.sumRange(0, 1);
// numArray.update(1, 10);
// numArray.sumRange(1, 2);
```

10.19 Range Sum Query 2D - Immutable (M)

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

Example:
Given matrix = [[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]
sumRegion(2, 1, 4, 3) = 8
sumRegion(1, 1, 2, 2) = 11
sumRegion(1, 2, 2, 4) = 12
Note:
You may assume that the matrix does not change.
There are many calls to sumRegion function.
You may assume that row1 $\leq$ row2 and col1 $\leq$ col2.

```
class NumMatrix {
public:
    int row, col;
    vector<vector<int>> sums;

    NumMatrix(vector<vector<int>> &matrix) {
        row = matrix.size();
        col = row > 0 ? matrix[0].size() : 0;
        sums = vector<vector<int>>(row+1, vector<int>(col+1, 0));
        for(int i = 1; i <= row; i++) {
            for(int j = 1; j <= col; j++) {
                sums[i][j] = sums[i-1][j] + sums[i][j-1] - sums[i-1][j-1] +
                    matrix[i-1][j-1];
            }
        }
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        return sums[row2+1][col2+1] - sums[row2+1][col1] - sums[row1][col2+1] +
            sums[row1][col1];
    }
};

// Your NumMatrix object will be instantiated and called as such:
// NumMatrix numMatrix(matrix);
// numMatrix.sumRegion(0, 1, 2, 3);
// numMatrix.sumRegion(1, 2, 3, 4);
```

10.20 Unique Paths (M)

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?

Note: m and n will be at most 100.

```
// 1. 2D DP solution
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m, vector<int>(n,1));
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
```

```
            }
        }
        return dp[m-1][n-1];
    }
};

// 2. 1D DP solution
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> dp(n,1);
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[j] = dp[j] + dp[j-1]; //
            }
        }
        return dp[n-1];
    }
};
```

10.21 Unique Paths II (M)

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.

```
// 1. 2D DP
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        if (obstacleGrid.empty() || obstacleGrid[0].empty()) return 0;
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) return 0;
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (obstacleGrid[i][j] == 1) {
                    dp[i][j] = 0;
                } else if (i == 0 && j == 0) {
                    dp[i][j] = 1;
```

```
                } else if (i == 0 && j > 0) {
                    dp[i][j] = dp[i][j-1];
                } else if (i > 0 && j == 0) {
                    dp[i][j] = dp[i-1][j];
                } else {
                    dp[i][j] = dp[i-1][j] + dp[i][j-1];
                }
            }
        }
        return dp[m-1][n-1];
    }
};

// 2. 1D DP
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        if (obstacleGrid.empty() || obstacleGrid[0].empty()) return 0;
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) return 0;
        vector<int> dp(n, 0);
        dp[0] = 1;
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (obstacleGrid[i][j] == 1) {
                    dp[j] = 0;
                } else if (j > 0) {
                    dp[j] = dp[j] + dp[j-1];
                }
            }
        }
        return dp[n-1];
    }
};
```

10.22 Minimum Path Sum (M)

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.

```
// 1. 2D DP
class Solution {
public:
```

```
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.size() == 0 || grid[0].size() == 0) return 0;
        int m = grid.size(), n = grid[0].size();
        int dp[m][n];
        // initializing
        dp[0][0] = grid[0][0];
        for (int i = 1; i < m; ++i) dp[i][0] = dp[i-1][0] + grid[i][0];
        for (int j = 1; j < n; ++j) dp[0][j] = dp[0][j-1] + grid[0][j];
        // get min
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1]);
            }
        }
        return dp[m-1][n-1];
    }
};

// 2. 1D DP
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.size() == 0 || grid[0].size() == 0) return 0;
        int m = grid.size(), n = grid[0].size();
        int dp[n];
        dp[0] = grid[0][0];
        for (int i = 1; i < n; ++i) dp[i] = dp[i-1] + grid[0][i];
        for (int i = 1; i < m; ++i) {
            dp[0] += grid[i][0]; // need to update dp[0] for each row
            for (int j = 1; j < n; ++j) {
                dp[j] = grid[i][j] + min(dp[j-1], dp[j]);
            }
        }
        return dp[n-1];
    }
};
```

10.23 Dungeon Game (M)

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.

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 m = dungeon.size(), n = dungeon[0].size();
        int dp[m][n];
        // Initialize K's health as K has to be alive when K reaches P
        // 1. If the current room can increase K's HP,
        // then the minimum HP for K to reach the room is 1
        // 2. If the current room can reduce K's HP,
        // then the minimum HP for K to reach the room must be 1 - damage
        dp[m-1][n-1] = max(1, 1 - dungeon[m-1][n-1]);
        // initializing the last column
        for (int i = m - 2; i >= 0; --i) {
            dp[i][n-1] = max(1, dp[i+1][n-1] - dungeon[i][n-1]);
        }
        // initializing the last row
        for (int j = n - 2; j >= 0; --j) {
            dp[m-1][j] = max(1, dp[m-1][j+1] - dungeon[m-1][j]);
        }
        for (int i = m - 2; i >= 0; --i) {
            for (int j = n - 2; j >= 0; --j) {
                dp[i][j] = max(1, min(dp[i+1][j], dp[i][j+1]) - dungeon[i][j]);
            }
        }
        return dp[0][0];
    }
};
```

10.24 Increasing Triplet Subsequence (M)

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Formally the function should:
Return true if there exists i, j, k
such that $arr[i] < arr[j] < arr[k]$ given $0 \leq i < j < k \leq n-1$ else return false.

Your algorithm should run in O(n) time complexity and O(1) space complexity.

Examples:
Given [1, 2, 3, 4, 5], return true.
Given [5, 4, 3, 2, 1], return false.

```
// 1. DP solution: Time O(N^2), Space O(N)
class Solution {
public:
    bool increasingTriplet(vector<int>& nums) {
        vector<int> dp(nums.size(), 1);
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                    if (dp[i] == 3) return true;
                }
            }
        }
        return false;
    }
};

// 2. Two minimum values: Time O(N), Space O(1)
class Solution {
public:
    bool increasingTriplet(vector<int>& nums) {
        int c1 = INT_MAX, c2 = INT_MAX;
        for (int i = 0; i < nums.size(); ++i) {
            if (nums[i] <= c1) c1 = nums[i]; // the first minimum
            else if (nums[i] <= c2) c2 = nums[i]; // the second minimum
            else return true; // the third value
        }
        return false;
    }
};
```

10.25 Longest Increasing Subsequence (M)

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example, Given [10, 9, 2, 5, 3, 7, 101, 18], The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

```
// 1. DP solution: Time O(N^2), Space O(N)
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> dp(nums.size(), 1);
        int res = 0;
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
            res = max(res, dp[i]);
        }
        return res;
    }
};

// 2. Binary search: Time O(NlogN), Space O(N)
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        if (nums.empty()) return 0;
        vector<int> res = {nums[0]};
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] < res[0]) { // update minimum
                res[0] = nums[i];
            } else if (nums[i] > res.back()) { // update maximum
                res.push_back(nums[i]);
            } else { // find values between the first and the last element of res
                and add them into res
                int left = 0, right = res.size() - 1;
                while (left < right) {
                    int mid = left + (right - left) / 2;
```

```
                if (res[mid] < nums[i]) left = mid + 1;
                else right = mid;
            }
            res[right] = nums[i];
        }
    }
    return res.size(); // the res size is the length of LIS
  }
};
```

10.26 Longest Increasing Path in a Matrix (M)

Given an integer matrix, find the length of the longest increasing path. From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:
nums = [[9,9,4], [6,6,8], [2,1,1]], Return 4, The longest increasing path is [1, 2, 6, 9].

Example 2:
nums = [[3,4,5], [3,2,6], [2,2,1]], Return 4, The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.

```
class Solution {
public:
    int longestIncreasingPath(vector<vector<int> >& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int res = 1, m = matrix.size(), n = matrix[0].size();
        vector<vector<int> > dp(m, vector<int>(n, 0));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                res = max(res, dfs(matrix, dp, i, j));
            }
        }
        return res;
    }
    int dfs(vector<vector<int> > &matrix, vector<vector<int> > &dp, int i, int j)
        {
        if (dp[i][j]) return dp[i][j];
        // move left, up, right, down
        vector<vector<int> > dirs = {{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
        int max_len = 1, m = matrix.size(), n = matrix[0].size();
        for (auto a : dirs) {
            int x = i + a[0], y = j + a[1]; // move (i,j) to next cell
```

```
            // avoid corner cases, look for the increasing path
            if (x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <=
                matrix[i][j]) continue;
            int len = 1 + dfs(matrix, dp, x, y);
            max_len = max(max_len, len);
        }
        dp[i][j] = max_len;
        return max_len;
    }
};
```

Chapter 11

Greedy Algorithm

11.1 Gas Station (M)

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.

Chapter 12

Graph

12.1 Number of Connected Components in an Undirected Graph (M)

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1: Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Example 2: Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1.

Note: You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

```
class Solution {
public:
    int countComponents(int n, vector<pair<int, int> >& edges) {
        vector<vector<int>> graph(n); // adjacency list
        vector<bool> visit(n, false); // visit state
        int res = 0;
        // create the adjacency list
        for (auto edge : edges) {
            graph[edge.first].push_back(edge.second);
            graph[edge.second].push_back(edge.first);
        }
        for (int i = 0; i < n; ++i) {
            if (!visit[i]) { // if i is not visited
                ++res; // increase the number of connected components
                dfs(graph, visit, i); // use dfs to visit all nodes in the current
                    conneted component
            }
        }
        return res;
```

```
    }

    void dfs(vector<vector<int>> &graph, vector<bool> &visit, int i) {
        if (visit[i]) return;
        visit[i] = true;
        for (int j = 0; j < graph[i].size(); ++j) {
            dfs(graph, visit, graph[i][j]); // traverse all nodes connected to i
        }
    }
};
```

12.2 Graph Valid Tree (M)

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:
Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.
Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Hint:
Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], what should your return? Is this case a valid tree?
According to the definition of tree on Wikipedia: ?a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.?

Note: you can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

```
// DFS
class Solution {
public:
    bool validTree(int n, vector<pair<int, int>>& edges) {
        vector<vector<int>> graph(n, vector<int>());
        vector<bool> visit(n, false);
        for (auto edge : edges) {
            g[edge.first].push_back(edge.second);
            g[edge.second].push_back(edge.first);
        }
        if (!dfs(graph, visit, 0, -1)) return false;
        for (auto v : visit) {
            if (!v) return false;
        }
```

```
        return true;
    }
    bool dfs(vector<vector<int>> &graph, vector<bool> &visit, int cur, int pre) {
        if (visit[cur]) return false;
        visit[cur] = true;
        for (auto g : graph[cur]) {
            if (g != pre) {
                if (!dfs(graph, visit, g, cur)) return false;
            }
        }
        return true;
    }
};

// BFS
class Solution {
public:
    bool validTree(int n, vector<pair<int, int>>& edges) {
        vector<unordered_set<int>> graph(n, unordered_set<int>());
        unordered_set<int> visit;
        queue<int> q;
        q.push(0);
        visit.insert(0);
        for (auto edge : edges) {
            graph[edge.first].insert(edge.second);
            graph[edge.second].insert(edge.first);
        }
        while (!q.empty()) {
            int t = q.front(); q.pop();
            for (auto g : graph[t]) {
                if (visit.find(g) != visit.end()) return false;
                visit.insert(g);
                q.push(g);
                graph[g].erase(t);
            }
        }
        return visit.size() == n;
    }
};
```

12.3 Clone Graph (M)

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

```
/**
```

```
 * Definition for undirected graph.
 * struct UndirectedGraphNode {
 *     int label;
 *     vector<UndirectedGraphNode *> neighbors;
 *     UndirectedGraphNode(int x) : label(x) {};
 * };
 */
class Solution {
public:
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
        // use hashtable to map label and neighbors
        unordered_map<int, UndirectedGraphNode*> umap;
        return clone(node, umap);
    }
    UndirectedGraphNode *clone(UndirectedGraphNode *node, unordered_map<int,
        UndirectedGraphNode*> &umap) {
        if (!node) return node;
        // if label is in the hashtable, return the list of neighbors
        if (umap.count(node->label)) return umap[node->label];
        // otherwise, define a new node that has label
        UndirectedGraphNode *newNode = new UndirectedGraphNode(node->label);
        // update hashtable
        umap[node->label] = newNode;
        // perform DFS
        // create a list of neighbors for the new node
        for (int i = 0; i < node->neighbors.size(); ++i) {
            (newNode->neighbors).push_back(clone(node->neighbors[i], umap));
        }
        return newNode;
    }
};
```

12.4 Number of Islands (M)

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) {
        if (grid.empty() || grid[0].empty()) return 0;
        int m = grid.size(), n = grid[0].size(), res = 0;
        vector<vector<bool> > visited(m, vector<bool>(n, false));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    numIslandsDFS(grid, visited, i, j);
                    ++res;
                }
            }
        }
        return res;
    }
    void numIslandsDFS(vector<vector<char> > &grid, vector<vector<bool> >
        &visited, int x, int y) {
        if (x < 0 || x >= grid.size()) return;
        if (y < 0 || y >= grid[0].size()) return;
        if (grid[x][y] != '1' || visited[x][y]) return;
        visited[x][y] = true; // grid[x][y] is visited now
        // search other 4 direction of grid[x][y]
        numIslandsDFS(grid, visited, x - 1, y);
        numIslandsDFS(grid, visited, x + 1, y);
        numIslandsDFS(grid, visited, x, y - 1);
        numIslandsDFS(grid, visited, x, y + 1);
    }
};
```

12.5 Number of Islands II (H)

A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. 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:
Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].

Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).
0 0 0
0 0 0
0 0 0

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.
1 0 0
0 0 0 Number of islands = 1
0 0 0

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.
1 1 0
0 0 0 Number of islands = 1
0 0 0

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.
1 1 0
0 0 1 Number of islands = 2
0 0 0

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.
1 1 0
0 0 1 Number of islands = 3
0 1 0

We return the result as an array: [1, 1, 2, 3]

Challenge: Can you do it in time complexity O(k log mn), where k is the length of the positions?

```
class Solution {
public:
    vector<int> numIslands2(int m, int n, vector<pair<int, int>>& positions) {
        vector<int> res;
        if (m <= 0 || n <= 0) return res;
        vector<int> roots(m * n, -1);
        int cnt = 0;
        vector<vector<int> > dirs{{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
        for (auto a : positions) {
            int id = n * a.first + a.second;
```

```
            roots[id] = id;
            ++cnt;
            for (auto d : dirs) {
                int x = a.first + d[0], y = a.second + d[1];
                int cur_id = n * x + y;
                if (x < 0 || x >= m || y < 0 || y >= n || roots[cur_id] == -1)
                    continue;
                int new_id = findRoots(roots, cur_id);
                if (id != new_id) {
                    roots[id] = new_id;
                    id = new_id;
                    --cnt;
                }
            }
            res.push_back(cnt);
        }
        return res;
    }
    int findRoots(vector<int> &roots, int id) {
        while (id != roots[id]) {
            roots[id] = roots[roots[id]];
            id = roots[id];
        }
        return id;
    }
};
```

Chapter 13

Details Implementation

13.1 Valid Sudoku (E)

Determine if a Sudoku is valid, according to:
1. Each row must have the numbers 1-9 occuring just once.
2. Each column must have the numbers 1-9 occuring just once.
3. And the numbers 1-9 must occur just once in each of the 9 sub-boxes of the grid.

The Sudoku board could be partially filled, where empty cells are filled with the character '.'. A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        bool used[9]; // count if the number in the cell is appeared

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

            // fill: Assigns val to all the elements in the range [first,last).
            // all set to false (not appear)
            fill(used, used + 9, false);
            // check rows
            for (int j = 0; j < 9; ++j) {
                if (!check(board[i][j], used))
                    return false;
            }

            fill(used, used + 9, false);
            // check columns
            for (int j = 0; j < 9; ++j) {
                if (!check(board[j][i], used))
                return false;
```

```
            }
        }

        // check 9 blocks
        for (int m = 0; m < 3; ++m) {
            for (int n = 0; n < 3; ++n) {
                fill(used, used + 9, false);
                for (int i = m * 3; i < m * 3 + 3; ++i) {
                    for (int j = n * 3; j < n * 3 + 3; ++j) {
                        if (!check(board[i][j], used))
                            return false;
                    }
                }
            }
        }

        return true;
    }

    bool check(char ch, bool used[9]) {
        if (ch == '.')    return true; // continue if the current cell is .
        if (used[ch - '1']) return false; // not first time appear, return false
        return used[ch - '1'] = true; // first time appear, return true
    }
};
```

13.2 Sudoku Solver (H)

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.

```
class Solution {
public:
    bool solveSudoku(vector<vector<char> > &board) {
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    for (int k = 0; k < 9; ++k) {
                        board[i][j] = '1' + k;
                        if (isValid(board, i, j) && solveSudoku(board))
                            return true;
                        board[i][j] = '.';
                    }
                    return false;
                }
```

```
            }
        }
        return true;
    }

    bool isValid(const vector<vector<char> > &board, int x, int y) {
        int i, j;
        for (i = 0; i < 9; i++) {
             if (i != x && board[i][y] == board[x][y])
                return false;
        }
        for (j = 0; j < 9; j++) {
               if (j != y && board[x][j] == board[x][y])
                 return false;
        }
        for (i = 3 * (x / 3); i < 3 * (x / 3 + 1); i++) {
            for (j = 3 * (y / 3); j < 3 * (y / 3 + 1); j++) {
                if ((i != x || j != y) && board[i][j] == board[x][y])
                   return false;
            }
        }
        return true;
    }
};
```

13.3 Design Tic-Tac-Toe (M)

Design a Tic-tac-toe game that is played between two players on a n x n grid.

You may assume the following rules:
A move is guaranteed to be valid and is placed on an empty block.
Once a winning condition is reached, no more moves is allowed.
A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:
Given n = 3, assume that player 1 is "X" and player 2 is "O" in the board.
TicTacToe toe = new TicTacToe(3);

toe.move(0, 0, 1); -¿ Returns 0 (no one wins)
—X— — —
— — — — // Player 1 makes a move at (0, 0).
— — — —

toe.move(0, 2, 2); -¿ Returns 0 (no one wins)

—X— —O—
— — — — // Player 2 makes a move at (0, 2).
— — — —

toe.move(2, 2, 1); -¿ Returns 0 (no one wins)
—X— —O—
— — — — // Player 1 makes a move at (2, 2).
— — —X—

toe.move(1, 1, 2); -¿ Returns 0 (no one wins)
—X— —O—
— —O— — // Player 2 makes a move at (1, 1).
— — —X—

toe.move(2, 0, 1); -¿ Returns 0 (no one wins)
—X— —O—
— —O— — // Player 1 makes a move at (2, 0).
—X— —X—

toe.move(1, 0, 2); -¿ Returns 0 (no one wins)
—X— —O—
—O—O— — // Player 2 makes a move at (1, 0).
—X— —X—

toe.move(2, 1, 1); -¿ Returns 1 (player 1 wins)
—X— —O—
—O—O— — // Player 1 makes a move at (2, 1).
—X—X—X—

Follow up:
Could you do better than $O(n^2)$ per move() operation?

Hint:
Could you trade extra space such that move() operation can be done in O(1)? You need two arrays: int rows[n], int cols[n], plus two variables: diagonal, antidiagonal.

```
class TicTacToe {
public:
    /** Initialize your data structure here. */
    TicTacToe(int n) {
        board.resize(n, vector<int>(n, 0));
    }

    int move(int row, int col, int player) {
```

```
        board[row][col] = player;
        int i = 0, j = 0, N = board.size();
        // check row
        for (i = 0; i < N; ++i) {
            if (board[i][0] != 0) {
                for (j = 1; j < N; ++j) {
                    if (board[i][j] != board[i][j - 1]) break;
                }
                if (j == N) return board[i][0];
            }
        }
        // check column
        for (j = 0; j < N; ++j) {
            if (board[0][j] != 0) {
                for (i = 1; i < N; ++i) {
                    if (board[i][j] != board[i - 1][j]) break;
                }
                if (i == N) return board[0][j];
            }
        }
        // check diagonal
        if (board[0][0] != 0) {
            for (i = 1; i < N; ++i) {
                if (board[i][i] != board[i - 1][i - 1]) break;
            }
            if (i == N) return board[0][0];
        }
        // check antidiagonal
        if (board[N - 1][0] != 0) {
            for (i = 1; i < N; ++i) {
                if (board[N - i - 1][i] != board[N - i][i - 1]) break;
            }
            if (i == N) return board[N - 1][0];
        }
        return 0;
    }

private:
    vector<vector<int>> board;
};
```

13.4 Boom Enemy (M)

Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb. The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is

too strong to be destroyed. Note that you can only put the bomb at an empty cell.

Example:
For the given grid
0 E 0 0
E 0 W E
0 E 0 0
return 3. (Placing a bomb at (1,1) kills 3 enemies)

```
class Solution {
public:
    int maxKilledEnemies(vector<vector<char>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        int m = grid.size(), n = grid[0].size(), res = 0, tmp;
        vector<vector<int>> v1(m, vector<int>(n, 0)), v2 = v1, v3 = v1, v4 = v1;
        // search each row
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) { // from left to right
                if (j == 0 || grid[i][j] == 'W') tmp = 0;
                else tmp = v1[i][j-1];
                if (grid[i][j] == 'E') v1[i][j] = tmp + 1;
                else v1[i][j] = tmp;
            }
            for (int j = n - 1; j >= 0; --j) { // from right to left
                if (j == n - 1 || grid[i][j] == 'W') tmp = 0;
                else tmp = v2[i][j+1];
                if (grid[i][j] == 'E') v2[i][j] = tmp + 1;
                else v2[i][j] = tmp;
            }
        }
        // search each column
        for (int j = 0; j < n; ++j) {
            for (int i = 0; i < m; ++i) { // from up to bottom
                if (i == 0 || grid[i][j] == 'W') tmp = 0;
                else tmp = v3[i-1][j];
                if (grid[i][j] == 'E') v3[i][j] = tmp + 1;
                else v3[i][j] = tmp;
            }
            for (int i = m - 1; i >= 0; --i) { // from bottom to up
                if (i == m - 1 || grid[i][j] == 'W') tmp = 0;
                else tmp = v4[i+1][j];
                if (grid[i][j] == 'E') v4[i][j] = tmp + 1;
                else v4[i][j] = tmp;
            }
        }
        // iterate all empty cell and get the max
```

```
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '0') {
                    res = max(res, v1[i][j] + v2[i][j] + v3[i][j] + v4[i][j]);
                }
            }
        }
        return res;
    }
};
```

13.5 Android Unlock Patterns (M)

Given an Android 3x3 key lock screen and two integers m and n, where $1 \leq m \leq n \leq 9$, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

Rules for a valid pattern:
Each pattern must connect at least m keys and at most n keys.
All the keys must be distinct.
If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
The order of keys used matters.

```
class Solution {
public:
    int numberOfPatterns(int m, int n) {
        int res = 0;
        vector<bool> visited(10, false);
        vector<vector<int>> jumps(10, vector<int>(10, 0));
        jumps[1][3] = jumps[3][1] = 2;
        jumps[4][6] = jumps[6][4] = 5;
        jumps[7][9] = jumps[9][7] = 8;
        jumps[1][7] = jumps[7][1] = 4;
        jumps[2][8] = jumps[8][2] = 5;
        jumps[3][9] = jumps[9][3] = 6;
        jumps[1][9] = jumps[9][1] = jumps[3][7] = jumps[7][3] = 5;
        res += helper(1, 1, 0, m, n, jumps, visited) * 4;
        res += helper(2, 1, 0, m, n, jumps, visited) * 4;
        res += helper(5, 1, 0, m, n, jumps, visited);
        return res;
    }
    int helper(int num, int len, int res, int m, int n, vector<vector<int>>
```

```
        &jumps, vector<bool> &visited) {
        if (len >= m) ++res;
        ++len;
        if (len > n) return res;
        visited[num] = true;
        for (int next = 1; next <= 9; ++next) {
            int jump = jumps[num][next];
            if (!visited[next] && (jump == 0 || visited[jump])) {
                res = helper(next, len, res, m, n, jumps, visited);
            }
        }
        visited[num] = false;
        return res;
    }
};
```

13.6 Game of Life (M)

According to the Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies, as if caused by under-population.
Any live cell with two or three live neighbors lives on to the next generation.
Any live cell with more than three live neighbors dies, as if by over-population.
Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:
Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

```
/** [2nd bit, 1st bit] = [next state, current state]
 * 00  dead (next) <- dead (current)
 * 01  dead (next) <- live (current)
 * 10  live (next) <- dead (current)
```

```
 * 11  live (next) <- live (current)
 *
 * 1. In the beginning, every cell is either 00 or 01.
 * 2. Notice that 1st state is independent of 2nd state.
 * 3. Imagine all cells are instantly changing from the 1st to the 2nd state, at
    the same time.
 * 4. Let's count # of neighbors from 1st state and set 2nd state bit.
 * 5. Since every 2nd state is by default dead, no need to consider transition
    01 -> 00.
 * 6. In the end, delete every cell's 1st state by doing >> 1.
 *
 * For each cell's 1st bit, check the 8 pixels around itself, and set the cell's
    2nd bit.
   Transition 01 -> 11: when board == 1 and (lives == 2 || lives == 3).
   Transition 00 -> 10: when board == 0 and lives == 3.
 * To get the current state, simply do board[i][j] & 1
 * To get the next state, simply do board[i][j] >> 1
 */
class Solution {
public:
    void gameOfLife(vector<vector<int>>& board) {
        if (board.empty() || board.size() == 0) return;
        int m = board.size(), n = board[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                int lives = getLives(board, i, j, m, n); // get current live
                    numbers
                if (board[i][j] == 1 && (lives == 2 || lives == 3)) {
                    board[i][j] = 3;          // 01 -> 11
                }

                if (board[i][j] == 0 && lives == 3) {
                    board[i][j] = 2;          // 00 -> 10
                }
            }
        }
        // update board based on the 2nd bit
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                board[i][j] >>= 1; // get the 2nd bit
            }
        }
    }
    int getLives(vector<vector<int>>& board, int i, int j, int m, int n) {
        int lives = 0;
        for (int p = max(i-1, 0); p <= min(i+1, m-1); ++p) {
            for (int q = max(j-1, 0); q <= min(j+1, n-1); ++q) {
```

```
                lives += board[p][q] & 1;    // add the 1st bit
            }
        }
        lives -= board[i][j] & 1;           // remove (i,j) itself
        return lives;
    }
};
```

Chapter 14

Brute-force Enumeration

Chapter 15

Divide and Conquer