INTRODUCTION TO ALGORITHMS

Top Coder

Help you to solve LeetCode's Question

Apply these fundamentals in a real interview

Top Coder

Help you to prepare technical coding interviews

Monster Chan

This book is for sale at http://leanpub.com/topcoder

This version was published on 2015-04-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Array **1**
1.1 Two Sum 1
1.2 3Sum 2
1.3 3Sum Closest 3
1.4 4Sum 4
1.5 Remove Element 9
1.6 Next Permutation 9
1.7 Permutation Sequence 11
1.8 Valid Sudoku 13
1.9 Trapping Rain Water 15
1.10 Rotate Image 17
1.11 Plus One 18
1.12 Word Search 20
1.13 Word Ladder II 22
1.14 Remove Duplicates from Sorted Array 25
1.15 Remove Duplicates from Sorted Array II 26
1.16 Search in Rotated Sorted Array 27
1.17 Search in Rotated Sorted Array II 28
1.18 Median of Two Sorted Arrays 29
1.19 Longest Consecutive Sequence 30
1.20 Set Matrix Zeroes 32
1.21 Merge Sorted Array 35
1.22 Merge Intervals 36
1.23 Rotate Array 37
1.24 Unique Paths 39
1.25 Unique Paths II 41
1.26 Triangle 43
1.27 Subsets 44
1.28 Subsets II 47
1.29 Spiral Matrix 52
1.30 Spiral Matrix II 53
1.31 Sort Colors 55
1.32 Pascal's Triangle 57

1.33 Pascal's Triangle II 58
1.34 Search Insert Position 59
1.35 Search for a Range 60
1.36 Search a 2D Matrix 61
1.37 Minimum Path Sum 62
1.38 Maximum Subarray 64
1.39 Maximum Product Subarray 66
1.40 Maximal Rectangle 67
1.41 Majority Element 68
1.42 Largest Rectangle in Histogram 68
1.43 Jump Game 69
1.44 Jump Game II 71
1.45 Insert Interval 72
1.46 First Missing Positive 75
1.47 Find Peak Element 76
1.48 Find Minimum in Rotated Sorted Array 77
1.49 Find Minimum in Rotated Sorted Array II 78
1.50 Container With Most Water 79
1.51 Construct Binary Tree from Preorder and Inorder Traversal 80
1.52 Construct Binary Tree from Inorder and Postorder Traversal 81
1.53 Combination Sum 82
1.54 Combination Sum II 82
1.55 Best Time to Buy and Sell Stock 84
1.56 Best Time to Buy and Sell Stock II 84
1.57 Best Time to Buy and Sell Stock III 85

1. Array

1.1 Two Sum

Given an array of integers, 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

Solution:
O(n^2) runtime, O(1) space – Brute force:

The brute force approach is simple. Loop through each element x and find if there is another value that equals to target – x. As finding another value requires looping through the rest of array, its runtime complexity is O(n^2), but it can be time out.

```
class Solution {
public:
    vector<int> twoSum(vector<int> &numbers, int target) {
        vector<int> result;
        for(int i = 0; i < numbers.size(); i++) {
            const int gap = target - numbers[i];
            for(int j = i+1; j < numbers.size(); j++) {
                if(numbers[j] == gap) {
                    result.push_back(i);
                    result.push_back(j);
                    break;
                }
            }
        }
        return result;
    }
};
```

O(n) runtime, O(n) space – Hash table:

We could reduce the runtime complexity of looking up a value to O(1) using a hash map that maps a value to its index.

```
class Solution {
public:
    vector<int> twoSum(vector<int> &numbers, int target) {
        unordered_map<int, int>mapping;
        vector<int> result;
        for (int i = 0; i < numbers.size(); i++) {
            mapping[numbers[i]] = i;
        }
        for (int i = 0; i < numbers.size(); i++) {
            const int gap = target - numbers[i];
            if (mapping.find(gap) != mapping.end() && mapping[gap] > i) {
                result.push_back(i + 1);
                result.push_back(mapping[gap] + 1);
                break;
            }
        }
        return result;
    }
};
```

1.2 3Sum

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

- Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)
- The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:

(-1, 0, 1)
(-1, -1, 2)

Solution:
O(n^2) runtime:

We could make a sort for the items and then make a squeeze.
This method can be extended to *k-Sum*.

```
class Solution {
public:
    vector<vector<int> > threeSum(vector<int> &num) {
        vector<vector<int>> result;
        if (num.size() < 3) return result;
        sort(num.begin(), num.end());
        const int target = 0;
        auto last = num.end();
        for(auto i = num.begin() ;i < last-2; i++) {
            auto j = i+1;
            if(i > num.begin() && *i == *(i-1)) continue;
            auto k = last -1;
            while(j < k) {
                if(*i + *j + *k < target) {
                    ++j;
                    while(*j == *(j-1) && j < k) ++j;
                }else if(*i + *j + *k > target) {
                    --k;
                    while(*k == *(k+1) && j < k) --k;
                }else {
                    result.push_back({*i, *j, *k});
                    ++j;
                    --k;
                    while(*j == *(j - 1) && *k == *(k+1) && j < k) ++j;
                }
            }
        }
        return result;
    }
};
```

1.3 3Sum Closest

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).

Solution:
$O(n^2)$ runtime

We could make a sort for the items and then make a squeeze.

```cpp
class Solution {
public:
    int threeSumClosest(vector<int> &num, int target) {
        int result;
        int min_gap = INT_MAX;

        sort(num.begin(), num.end());
        for(auto a = num.begin(); a != prev(num.end(), 2); ++a) {
            auto b = next(a);
            auto c = prev(num.end());

            while(b < c) {
                const int sum = *a + *b + *c;
                const int gap = abs(sum - target);

                if(gap < min_gap) {
                    result = sum;
                    min_gap = gap;
                }

                if(sum < target) ++b;
                else --c;
            }
        }
        return result;
    }
};
```

1.4 4Sum

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.

Note:

- Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d)
- 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)

Solution:
O(n^3) runtime, O(1) space:

We could make a sort for the items and then make a squeeze.

```
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {
        vector<vector<int>> result;
        if(num.size() < 4){
            return result;
        }

        sort(num.begin(), num.end());
        auto last = num.end();

        for(auto a = num.begin(); a < prev(num.end(), 3); ++a) {
            for(auto b = next(a); b < prev(num.end(), 2); ++b) {
                auto c = next(b);
                auto d = prev(last);

                while(c < d ) {
                    if(*a + *b + *c + *d < target) {
                        ++c;
                    }else if(*a + *b + *c + *d > target) {
                        --d;
                    }else {
                        result.push_back({*a, *b, *c, *d});
                        ++c;
                        --d;
                    }
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

O(n^3logn) runtime, O(1) space:

Other way, it looks more better, in fact is was more slowly and could be time out.

```
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {
        vector<vector<int>> result;
        if(num.size() < 4){
            return result;
        }

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

        auto last = num.end();
        for(auto a = num.begin(); a < prev(last, 3); a++) {
            a = upper_bound(a, prev(last, 3), *a);
            for(auto b = next(a); b < prev(last, 2); b++) {
                b= upper_bound(b, prev(last, 2), *b);
                auto c = next(b);
                auto d = prev(last);
                while(c < d) {
                    if(*a + *b + *c + *d < target) {
                        c = upper_bound(c, d, *c);
                    }else if(*a + *b + *c + *d > target) {
                        d = prev(lower_bound(c, d, *d));
                    }else {
                        result.push_back({*a, *b, *c, *d});
                        c = upper_bound(c, d, *c);
                        d = prev(lower_bound(c, d, *d));
                    }
                }
            }
        }
        return result;
    }
};
```

$O(n^2)$ average runtime, $O(n^4)$ worst runtime, $O(n^2)$ space:

We will use hashmap as the cache for the two number.

```
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {
        vector<vector<int>> result;
        if(num.size() < 4){
            return result;
        }

        sort(num.begin(), num.end());
        unordered_map<int, vector<pair<int, int>>> cache;

        for(size_t a = 0; a < num.size(); a++) {
            for(int b = a + 1; b < num.size(); b++) {
                cache[num[a]+num[b]].push_back(pair<int, int>(a, b));
            }
        }

        for(int c = 0; c < num.size(); c++) {
            for(int d = c + 1; d < num.size(); d++) {
                const int key = target - num[c] - num[d];
                if(cache.find(key) == cache.end()) {
                    continue;
                }

                const auto &vec = cache[key];
                for(int k = 0; k < vec.size(); k++) {
                    if(c <= vec[k].second) {
                        continue;
                    }
                    result.push_back({num[vec[k].first], num[vec[k].second], num\
[c], num[d]});
                }
            }
        }

        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

$O(n^2)$ runtime, $O(n^2)$ space:

Improved algorithm using multimap.

```
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {
        vector<vector<int>> result;
        if(num.size() < 4){
            return result;
        }

        sort(num.begin(), num.end());
        unordered_multimap<int, pair<int, int>> cache;

        for(int a = 0; a + 1 < num.size(); a++) {
            for(int b = a + 1; b < num.size(); b++) {
                cache.insert(make_pair(num[a]+num[b], make_pair(a, b)));
            }
        }

        for(auto i = cache.begin(); i != cache.end(); i++) {
            int gap = target - i->first;
            auto range = cache.equal_range(gap);
            for(auto j = range.first; j != range.second; j++) {
                auto a = i->second.first;
                auto b = i->second.second;
                auto c = j->second.first;
                auto d = j->second.second;

                if(a != c && a != d && b != c && b != d) {
                    vector<int> vec = {num[a], num[b], num[c], num[d]};
                    sort(vec.begin(), vec.end());
                    result.push_back(vec);
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

1.5 Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Solution:
O(n) runtime, O(1) space:

```
class Solution {
public:
    int removeElement(int A[], int n, int elem) {
        int index = 0;
        for(int i = 0; i < n; i++) {
            if(A[i] != elem){
                A[index++] = A[i];
            }
        }
        return index;
    }
};
```

```
class Solution {
public:
    int removeElement(int A[], int n, int elem) {
        return distance(A, remove(A, A+n, elem));
    }
};
```

1.6 Next Permutation

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

Solution:
O(n) runtime, O(1) space:

1. From right to left, find the first digit (PartitionNumber) which violate the increase trend.
2. From right to left, find the first digit which large than PartitionNumber, call it changeNumber.
3. Swap the PartitionNumber and ChangeNumber.
4. Reverse all the digit on the right og partition index.

```
class Solution {
public:
    void nextPermutation(vector<int> &num) {
        next_permutation(num.begin(), num.end());
    }

    template<typename BidiIt>
    bool next_permutation(BidiIt first, BidiIt last) {

        // Get a reversed range to simplify reversed traversal.
        const auto rfirst = reverse_iterator<BidiIt>(last);
        const auto rlast = reverse_iterator<BidiIt>(first);

        // Begin from the second last element to the first element.
        auto pivot = next(rfirst);

        // Find `pivot`, which is the first element that is no less than its
        // successor.  `Prev` is used since `pivort` is a `reversed_iterator`.
        while (pivot != rlast && *pivot >= *prev(pivot))
            ++pivot;

        if (pivot == rlast) {
            reverse(rfirst, rlast);
            return false;
        }

        auto change = find_if(rfirst, pivot, bind1st(less<int>(), *pivot));

        swap(*change, *pivot);
        reverse(rfirst, pivot);
```

```
31
32          return true;
33
34      }
35  };
```

1.7 Permutation Sequence

The set [1,2,3,...,n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order,

We get the following sequence (ie, for n = 3):

“123”
“132”
“213”
“231”
“312”
“321”

Given *n* and *k*, return the k^{th} permutation sequence.

Note: Given *n* will be between 1 and 9 inclusive.

Solution:
Brute force, call k-1 times next_permutation(), but that may be time out.

```
1   class Solution {
2   public:
3       string getPermutation(int n, int k) {
4
5           string s(n, '0');
6           for(int i = 0; i < n; i++) {
7               s[i] += i+1;
8           }
9           for(int i = 0; i < k-1; i++) {
10              next_permutation(s.begin(), s.end());
11          }
12          return s;
13      }
14
15      template<typename BidiIt>
16      bool next_permutation(BidiIt first, BidiIt last) {
17
```

```
        // Get a reversed range to simplify reversed traversal.
        const auto rfirst = reverse_iterator<BidiIt>(last);
        const auto rlast = reverse_iterator<BidiIt>(first);

        // Begin from the second last element to the first element.
        auto pivot = next(rfirst);

        // Find `pivot`, which is the first element that is no less than its
        // successor.  `Prev` is used since `pivort` is a `reversed_iterator`.
        while (pivot != rlast && *pivot >= *prev(pivot))
            ++pivot;

        if (pivot == rlast) {
            reverse(rfirst, rlast);
            return false;
        }

        auto change = find_if(rfirst, pivot, bind1st(less<int>(), *pivot));

        swap(*change, *pivot);
        reverse(rfirst, pivot);

        return true;

    }
};
```

Cantor expansion, O(n) runtime, O(1) space:

```
class Solution {
public:
    string getPermutation(int n, int k) {

        string s(n, '0');
        string result;
        for(int i = 0; i < n; ++i) {
            s[i] += i+1;
        }
        return kth_permutation(s, k);
    }

private:
```

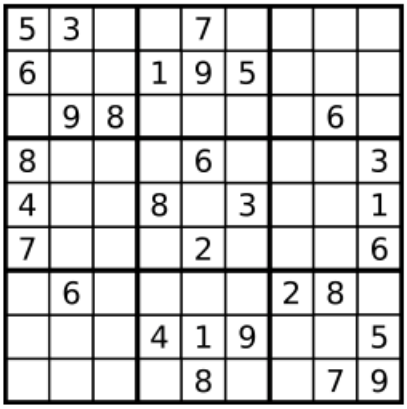
```
14      int factorial(int n) {
15          int result = 1;
16          for(int i = 1; i <= n; i++) {
17              result *= i;
18          }
19          return result;
20      }
21
22      template<typename Sequence>
23      Sequence kth_permutation(const Sequence &seq, int k) {
24          const int n = seq.size();
25          Sequence S(seq);
26          Sequence result;
27          int base = factorial(n - 1);
28          --k;
29          for (int i = n - 1; i > 0; k %= base, base /= i, --i) {
30                auto a = next(S.begin(), k / base);
31                result.push_back(*a);
32                S.erase(a);
33          }
34          result.push_back(S[0]);
35            return result;
36        }
37  };
```

1.8 Valid Sudoku

Determine if a Sudoku is valid, according to:Sudoku Puzzles - The Rules.[1]

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

[1] http://sudoku.com.au/TheRules.aspx



A partially filled sudoku which is valid.

Note:
A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

Solution:
O(n^2) runtime, O(1) space:

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char> > &board) {
        const int cnt = 9;
        bool row_mask[cnt][cnt] = {false};
        bool col_mask[cnt][cnt] = {false};
        bool area_mask[cnt][cnt] = {false};

        for(int r = 0; r < board.size(); r++) {
            for(int c = 0; c < board[r].size(); c++) {
                if(!isdigit(board[r][c])) continue;

                int idx = board[r][c] - '0' - 1;

                if(row_mask[r][idx] == true) {
                    return false;
                }

                row_mask[r][idx] = true;

                if (col_mask[c][idx] == true) {
                    return false;
                }
```

```
24                  col_mask[c][idx] = true;
25
26                  int area = (r/3) * 3 + (c/3);
27                  if (area_mask[area][idx] == true) {
28                      return false;
29                  }
30                  area_mask[area][idx] = true;
31              }
32          }
33          return true;
34      }
35  };
```

1.9 Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

Solution:
O(n) runtime, **O(1) space**:
1. find the highest bar.
2. traverse the bar from left the highest bar.becasue we have the highest bar in right, so, any bar higher than its right bar(s) can contain the water.
3. traverse the bar from right the highest bar.becasue we have the highest bar in left, so, any bar higher than its left bar(s) can contain the water.

```
class Solution {
public:
    int trap(int A[], int n) {
        int max = 0;
        for(int i=0; i< n; i++){
            if(A[i] > A[max]) {
                max = i;
            }
        }

        int water = 0;
        for(int i=0, peak=0; i<max; i++) {
            if(A[i] > peak){
                peak = A[i];
            }else {
                water += peak - A[i];
            }
        }

        for(int i=n-1, top=0; i>max; i--) {
            if(A[i] > top) {
                top = A[i];
            }else {
                water += top - A[i];
            }
        }

        return water;
    }
};
```

O(n) runtime, O(n) space:
1. scan from left to right, for each column, find the maximum value to its left.
2. scan from right to left, for each column, find the maximun value to its right.
3. Then scan again, the cumulative area of each column.

```
class Solution {
public:
    int trap(int A[], int n) {
        int *max_left = new int[n]();
        int *max_right = new int[n]();

        for(int i=1; i<n; i++) {
            max_left[i] = max(max_left[i - 1], A[i - 1]);
            max_right[n - 1 - i] = max(max_right[n - i], A[n - i]);
        }

        int sum = 0;
        for(int i=0; i<n; i++) {
            int height = min(max_left[i], max_right[i]);
            if(height > A[i]) {
                sum += height - A[i];
            }
        }

        delete[] max_left;
        delete[] max_right;
        return sum;
    }
};
```

1.10 Rotate Image

You are given an *n x n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:

Could you do this in-place?

Solution:
O(n^2) runtime, O(1) space:

```cpp
class Solution {
public:
    void rotate(vector<vector<int> > &matrix) {
        const int n = matrix.size();

        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]);
            }
        }

        for(int i = 0; i < n/2; ++i) {
            for (int j = 0; j < n; ++j) {
                swap(matrix[i][j], matrix[n - 1 - i][j]);
            }
        }
    }
};
```

$O(n^2)$ runtime, O(1) space:

```cpp
class Solution {
public:
    void rotate(vector<vector<int> > &matrix) {
        const int n = matrix.size();

        for(int i = 0; i < n / 2; i++) {
            for(int j = 0; j < n; j++) {
                swap(matrix[i][j], matrix[n - 1 - i][j]);
            }
        }

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

1.11 Plus One

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.

Solution:
O(n) runtime, O(1) space:

```
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
        return digits;
    }
private:
    void add(vector<int> &digits, int digit) {
        int c = digit;

        for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
            *it += c;
            c = *it / 10;
            *it %= 10;

        }
        if (c > 0) digits.insert(digits.begin(), 1);
    }
};
```

O(n) runtime, O(1) space:

```
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
        return digits;
    }
private:
    void add(vector<int> &digits, int digit) {
        int c = digit;

        for_each(digits.rbegin(), digits.rend(), [&c](int &d){
            d += c;
            c = d / 10;
            d %= 10;

        });
```

```
        if (c > 0) digits.insert(digits.begin(), 1);
    }
};
```

1.12 Word Search

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

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

For example,

Given board =

[
["ABCE"],
["SFCS"],
["ADEE"]
]

word = "ABCCED", -> returns true,

word = "SEE", -> returns true,

word = "ABCB", -> returns false.

Solution:

```
class Solution {
public:

    bool exist(vector<vector<char> > &board, string word, int idx, int row, int \
col, vector< vector<int> > &mask) {
        int i = row;
        int j = col;
        if (board[i][j] == word[idx] && mask[i][j]==0 ) {
            mask[i][j]=1; //mark the current char is matched
            if (idx+1 == word.size()) return true;
            //checking the next char in `word` through the right, left, up, down\
 four directions in the `board`.
            idx++;
            if (( i+1<board.size()    && exist(board, word, idx, i+1, j, mask) )\
 ||
                ( i>0                 && exist(board, word, idx, i-1, j, mask) )\
 ||
```

```
                ( j+1<board[i].size() && exist(board, word, idx, i, j+1, mask) )\
 ||
                ( j>0                 && exist(board, word, idx, i, j-1, mask) )\
 )
            {
                return true;
            }
            mask[i][j]=0; //cannot find any successful solution, clear the mark.\
 (backtracking)
        }

        return false;
    }

    vector< vector<char> > buildBoard(char b[][5], int r, int c) {
        vector< vector<char> > board;
        for (int i=0; i<r; i++){
            vector<char> v(b[i], b[i]+c);
            cout << b[i] << endl;
            board.push_back(v);
        }
        //cout << "----------" << endl;
        return board;
    }

    bool exist(vector<vector<char> > &board, string word) {
        if (board.size()<=0 || word.size()<=0) return false;
        int row = board.size();
        int col = board[0].size();
        //using a mask to mark which char has been selected.
        //do not use vector<bool>, it has big performance issue, could cause Tim\
e Limit Error
        vector< vector<int> > mask(row, vector<int>(col, 0));

        for(int i=0; i<board.size(); i++) {
            for(int j=0; j<board[i].size(); j++){
                if ( board[i][j]==word[0] ){
                    vector< vector<int> > m = mask;
                    if( exist(board, word, 0, i, j, m) ){
                        return true;
                    }
                }
```

```
60                 }
61             }
62             return false;
63         }
64 };
```

1.13 Word Ladder II

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

1.Only one letter can be changed at a time

2.Each intermediate word must exist in the dictionary

For example,

Given:

start = “hit”

end = “cog”

dict = [“hot”,”dot”,”dog”,”lot”,”log”]

Return

[
[“hit”,”hot”,”dot”,”dog”,”cog”],
[“hit”,”hot”,”lot”,”log”,”cog”]
]

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

Solution:

- Using BSF algorithm build a tree
- Using DSF to parse the tree to the transformation path

```
class Solution {
public:

    map< string, unordered_set<string> >&
    buildTree(string& start, string& end, unordered_set<string> &dict) {
        static map< string, unordered_set<string> > parents;
        parents.clear();

        unordered_set<string> level[3];
        unordered_set<string> *previousLevel = &level[0];
        unordered_set<string> *currentLevel = &level[1];
        unordered_set<string> *newLevel = &level[2];
        unordered_set<string> *p =NULL;
        currentLevel->insert(start);

        bool reachEnd = false;

        while( !reachEnd ) {
            newLevel->clear();
            for(auto it=currentLevel->begin(); it!=currentLevel->end(); it++) { \

                for (int i=0; i<it->size(); i++) {
                    string newWord = *it;
                    for(char c='a'; c<='z'; c++){
                        newWord[i] = c;
                        if (newWord == end){
                            reachEnd = true;
                            parents[*it].insert(end);
                            continue;
                        }
                        if ( dict.count(newWord)==0 || currentLevel->count(newWo\
rd)>0 || previousLevel->count(newWord)>0 ) {
                            continue;
                        }
                        newLevel->insert(newWord);
                        //parents[newWord].insert(*it);
                        parents[*it].insert(newWord);
                    }
                }
            }
            if (newLevel->empty()) break;

```

```
43              p = previousLevel;
44              previousLevel = currentLevel;
45              currentLevel = newLevel;
46              newLevel = p;
47          }
48
49
50          if ( !reachEnd ) {
51              parents.clear();
52          }
53          return parents;
54      }
55
56      void generatePath( string start, string end,
57          map< string, unordered_set<string> > &parents,
58          vector<string> path,
59          vector< vector<string> > &paths) {
60
61          if (parents.find(start) == parents.end()){
62              if (start == end){
63                  paths.push_back(path);
64              }
65              return;
66          }
67
68          for(auto it=parents[start].begin(); it!=parents[start].end(); it++){
69              path.push_back(*it);
70              generatePath(*it, end, parents, path, paths);
71              path.pop_back();
72          }
73
74      }
75      vector<vector<string>> findLadders(string start, string end, unordered_set<s\
76  tring> &dict) {
77
78          vector< vector<string> > ladders;
79          vector<string> ladder;
80          ladder.push_back(start);
81          if (start == end){
82              ladder.push_back(end);
83              ladders.push_back(ladder);
84              return ladders;
```

```
85          }
86
87          map< string, unordered_set<string> >& parents = buildTree(start, end, di\
88  ct);
89
90          if  (parents.size()<=0) {
91              return ladders;
92          }
93
94          generatePath(start, end, parents, ladder, ladders);
95
96          return ladders;
97      }
98  };
```

1.14 Remove Duplicates from Sorted Array

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 A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

Solution:
O(n) runtime, O(1) space:

```
1   class Solution {
2   public:
3       int removeDuplicates(int A[], int n) {
4
5           if(n == 0) {
6               return 0;
7           }
8
9           int index = 0;
10          for(int i = 0; i < n; i++) {
11              if(A[index] != A[i]) {
12                  A[++index] = A[i];
13              }
```

```
        }
        return index + 1;
    }
};
```

O(n) runtime, O(1) space:

<<(code/RemoveDuplicatesfromSortedArray2.cpp)

O(n) runtime, O(1) space:

```
class Solution {
public:
    int removeDuplicates(int A[], int n) {

        return removeDuplicates(A, A + n, A) - A;
    }

    template<typename InIt, typename OutIt>
    OutIt removeDuplicates(InIt first, InIt last, OutIt output) {
        while(first != last) {
            *output++ = *first;
            first = upper_bound(first, last, *first);
        }
        return output;
    }
};
```

1.15 Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":

What if duplicates are allowed at most *twice*?

For example,

Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].

Solution:
O(n) runtime, O(1) space:

```
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        if(n <= 2) {
            return n;
        }
        int index = 2;
        for(int i = 2; i < n; i++) {
            if(A[i] != A[index- 2]) {
                A[index++] = A[i];
            }
        }
        return index;
    }
};
```

O(n) runtime, O(1) space:

```
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        int index = 0;
          for (int i = 0; i < n; ++i) {
              if (i > 0 && i < n - 1 && A[i] == A[i - 1] && A[i] == A[i + 1])
                  continue;
              A[index++] = A[i];
          }
          return index;
    }
};
```

1.16 Search in Rotated Sorted Array

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

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

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Solution:
O(logn) runtime, O(1) space:

```
class Solution {
public:
    int search(int A[], int n, int target) {
        int first = 0, last = n;
        while (first != last) {
            const int mid = first  + (last - first) / 2;
            if (A[mid] == target){
                return mid;
            }
            if (A[first] <= A[mid]) {
                if (A[first] <= target && target < A[mid])
                     last = mid;
                  else
                      first = mid + 1;
            } else {
                  if (A[mid] < target && target <= A[last-1])
                      first = mid + 1;
                  else
                      last = mid;
            }
        }
        return -1;
    }
};
```

1.17 Search in Rotated Sorted Array II

Follow up for “Search in Rotated Sorted Array”:

What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Solution:
O(n) runtime, O(1) space:

```
class Solution {
public:
    bool search(int A[], int n, int target) {
        int first = 0, last = n;
          while (first != last) {
              const int mid = first  + (last - first) / 2;
              if (A[mid] == target)
                  return true;
              if (A[first] < A[mid]) {
                  if (A[first] <= target && target < A[mid])
                      last = mid;
                  else
                      first = mid + 1;
              } else if (A[first] > A[mid]) {
                  if (A[mid] < target && target <= A[last-1])
                      first = mid + 1;
                  else
                      last = mid;
              } else
                  //skip duplicate one
                  first++;
           }
          return false;
    }
};
```

1.18 Median of Two Sorted Arrays

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Solution:
O(log(m+n)) runtime, O(log(m+n)) space:

```
class Solution {
public:
    double findMedianSortedArrays(int A[], int m, int B[], int n) {

        int total = m + n;
        if (total & 0x1)
            return find_kth(A, m, B, n, total / 2 + 1);
        else
            return (find_kth(A, m, B, n, total / 2) + find_kth(A, m, B, n, total\
 / 2 + 1)) / 2.0;
    }

private:
    static int find_kth(int A[], int m, int B[], int n, int k) {

        if (m > n) return find_kth(B, n, A, m, k);
        if (m == 0) return B[k - 1];
        if (k == 1) return min(A[0], B[0]);

        int ia = min(k / 2, m), ib = k - ia;
        if (A[ia - 1] < B[ib - 1])
            return find_kth(A + ia, m - ia, B, n, k - ia);
        else if(A[ia - 1] > B[ib - 1])
            return find_kth(A, m, B + ib, n - ib, k - ib);
        else
            return A[ia - 1];
    }
};
```

1.19 Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity.

Solution:

Obviously, the easist way is sort the array, however the run-time complexity is O(nlogn)

If we cannot use the sort algorithm, then it seems we have to use $O(n^2)$ solution.

That's fine, let's take a look the $O(n^2)$ solution:

1) for each item num[i] in the array

2) for loop to search ..... num[i-2], num[i-1], num[i]+1, num[i]+2 .....

we can see the search is really heavy, and the beat data structure for searching is HashMap. Hash map is O(1) runtime complexity for seaching.

So, we can have the following solution by using Hash Map.

O(n) runtime, O(n) space:

```
class Solution {
public:
    int longestConsecutive(vector<int> &num) {
        unordered_map<int, bool> used;

        for (auto i : num) used[i] = false;

        int longest = 0;

        for (auto i : num) {
            if (used[i]) continue;

            int length = 1;

            used[i] = true;

            for (int j = i + 1; used.find(j) != used.end(); ++j) {
                used[j] = true;
                ++length;
            }

            for (int j = i - 1; used.find(j) != used.end(); --j) {
                used[j] = true;
                ++length;
            }
            longest = max(longest, length);
        }
        return longest;
    }
};
```

O(n) runtime, O(n) space:

```cpp
class Solution {
public:
    int longestConsecutive(vector<int> &num) {
        unordered_map<int, int> map;
        int size = num.size();
        int l = 1;

        for (int i = 0; i < size; i++) {
            if (map.find(num[i]) != map.end()) continue;
            map[num[i]] = 1;

            if (map.find(num[i] - 1) != map.end()) {
                l = max(l, mergeCluster(map, num[i] - 1, num[i]));
            }

            if (map.find(num[i] + 1) != map.end()) {
                l = max(l, mergeCluster(map, num[i], num[i] + 1));
            }
        }
        return size == 0 ? 0 : l;
    }

private:
    int mergeCluster(unordered_map<int, int> &map, int left, int right) {
        int upper = right + map[right] - 1;
        int lower = left - map[left] + 1;
        int length = upper - lower + 1;
        map[upper] = length;
        map[lower] = length;

        return length;
    }
};
```

1.20 Set Matrix Zeroes

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

Follow up: Did you use extra space?

A straight forward solution using *O(mn)* space is probably a bad idea.

A simple improvement uses *O(m + n)* space, but still not the best solution.

Could you devise a constant space solution?

Solution:
O(m*m) runtime, O(n+m) space:

```
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        const size_t n = matrix[0].size();
        vector<bool> row(m, false);
        vector<bool> col(n, false);

        for (size_t i = 0; i < m; ++i) {
            for (size_t j = 0; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    row[i] = col[j] = true;
                }
            }
        }

        for (size_t i = 0; i < m; ++i) {
            if (row[i]) {
                fill(&matrix[i][0], &matrix[i][0] + n, 0);
            }
        }

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

O(m*m) runtime, O(1) space:

```cpp
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        const size_t n = matrix[0].size();
        bool row_has_zero = false;
        bool col_has_zero = false;

        for(size_t i = 0; i < n; i++) {
            if(matrix[0][i] == 0) {
                row_has_zero = true;
                break;
            }
        }

        for(size_t i = 0; i< m; i++) {
            if (matrix[i][0] == 0) {
                col_has_zero = true;
                break;
            }
        }

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

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

        if (row_has_zero) {
            for (size_t i = 0; i < n; i++) {
                matrix[0][i] = 0;
```

```
            }
        }

        if (col_has_zero) {
            for (size_t i = 0; i < m; i++) {
                matrix[i][0] = 0;
            }
        }
    }
};
```

1.21 Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note:

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

Solution:
O(m+n) runtime, O(1) space:

```
class Solution {
public:
    void merge(int A[], int m, int B[], int n) {
        int ia = m-1 ;
        int ib = n-1 ;
        int i = m + n - 1;
        for (int i=m+n-1; i>=0; i--){

            if (ia>=0 && ib<0){
                break;
            }
            if (ia<0 && ib>=0){
                A[i] = B[ib--];
                continue;
            }
            if (ia>=0 && ib>=0){
                if (A[ia] > B[ib]){
                    A[i] = A[ia--];
                }else{
                    A[i] = B[ib--];
```

```
                }
            }

        }
    }
};
```

1.22 Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,

Given [1,3],[2,6],[8,10],[15,18],

return [1,6],[8,10],[15,18].

Solution:

```
/**
 * Definition for an interval.
 * struct Interval {
 *     int start;
 *     int end;
 *     Interval() : start(0), end(0) {}
 *     Interval(int s, int e) : start(s), end(e) {}
 * };
 */

bool compare(const Interval& lhs, const Interval& rhs){
    return (lhs.start==rhs.start) ? lhs.end < rhs.end : lhs.start < rhs.start;
}

class Solution {
public:


    vector<Interval> merge(vector<Interval> &intervals) {
        vector<Interval> result;

        if (intervals.size() <= 0) return result;

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

```

```
        for(int i=0; i<intervals.size(); i++) {
            int size = result.size();
            if( size>0 && result[size-1].end >= intervals[i].start) {
                result[size-1].end = max(result[size-1].end, intervals[i].end);
            }else {
                result.push_back(intervals[i]);
            }
        }
        return result;
    }
};
```

1.23 Rotate Array

Rotate an array of n elements to the right by k steps.

For example, with $n = 7$ and $k = 3$, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

Solution:

How to change [0,1,2,3,4,5,6] to [4,5,6,0,1,2,3] by k = 3?

We can change by following rules:

[0]->[3], [3]->[6], [6]->[2], [2]->[5], [5]->[1], [1]->[4]

```
class Solution {
public:
    void rotate(int nums[], int n, int k) {
        if (k<=0) return;
        k %= n;
        int currIdx=0, newIdx=k;
        int tmp1 = nums[currIdx], tmp2;
        int origin = 0;

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

            tmp2 = nums[newIdx];
            nums[newIdx] = tmp1;
            tmp1 = tmp2;

            currIdx = newIdx;
```

```
17
18              //if we meet a circle, move the next one
19              if (origin == currIdx) {
20                  origin = ++currIdx;
21                  tmp1 = nums[currIdx];
22              }
23          newIdx = (currIdx + k) % n;
24
25          }
26      }
27  };
```

This solution is so-called three times rotate method.

Because $(X^{TY}T)^{T} = YX$, so we can perform rotate operation three times to get the result.

Obviously, the algorithm consumes O(1) space and O(n) time.

```
1   class Solution {
2   public:
3       void reverseArray(int nums[],int start, int end){
4           int temp;
5           while(start < end){
6               int temp = nums[start];
7               nums[start++] = nums[end];
8               nums[end--] = temp;
9           }
10      }
11
12      void rotate1(int nums[], int n, int k) {
13          if (k<=0) return;
14          k %= n;
15          reverseArray(nums, n-k, n-1);
16          reverseArray(nums, 0, n-k-1);
17          reverseArray(nums, 0, n-1);
18      }
19
20      void rotate(int nums[], int n, int k) {
21          if (random()%2==0) {
22              return rotate1(nums, n, k);
23          }
24          return rotate1(nums, n, k);
25      }
26  };
```

1.24 Unique Paths

A robot is located at the top-left corner of a *m x n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Note: *m* and *n* will be at most 100.

Solution:
Deep Search, small collection can pass, a large collection will timeout.
$O(n^4)$ runtime, o(n) space:

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        if(m<1||n<1) return 0;
        if(m==1&&n==1) return 1;

        return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
    }
};
```

Deep Search with Cache $O(n^2)$ runtime, $O(n^2)$ space:

```cpp
class Solution {
public:
    int uniquePaths(int m, int n) {

        this->f = vector<vector<int> >(m + 1, vector<int>(n + 1, 0));
        return dfs(m, n);
    }
private:
    vector<vector<int> > f;

    int dfs(int x, int y) {
        if(x < 1 || y < 1) return 0;
        if (x == 1 && y == 1) return 1;

        return getOrUpdate(x - 1, y) + getOrUpdate(x, y - 1);
    }

    int getOrUpdate(int x, int y) {
        if (f[x][y] > 0) return f[x][y];
        else return f[x][y] = dfs(x, y);
    }
};
```

Dynamic Programming
State transition equation: f[i][j]=f[i-1][j]+f[i][j-1]
$O(n^2)$ runtime, O(n) space:

```cpp
class Solution {
public:
    int uniquePaths(int m, int n) {

        vector<int> f(n, 0);

        f[0] = 1;

        for (int i = 0; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[j] = f[j - 1] + f[j];
            }
        }
        return f[n - 1];
    }
```

```
16
17 };
```

Mathematical formula
A *m* rows and *n* columns of the matrix, the number of steps walked robot left on the lower right is the total of *m + n - 2*, wherein the number of steps to go down is *m - 1*, so the problem becomes the *m + n - 2* operation.
(m+n-2)!/((m-1)!(m+n-2-m+1)!)

```
1  class Solution {
2  public:
3
4      typedef long long int64_t;
5
6      static int64_t factor(int n, int start = 1) {
7          int64_t  ret = 1;
8          for(int i = start; i <= n; ++i)
9              ret *= i;
10             return ret;
11     }
12
13     static int64_t combination(int n, int k) {
14         if (k == 0) return 1;
15         if (k == 1) return n;
16
17         int64_t ret = factor(n, k+1);
18         ret /= factor(n - k);
19         return ret;
20     }
21
22     int uniquePaths(int m, int n) {
23         return combination(m+n-2, max(m-1, n-1));
24     }
25
26 };
```

1.25 Unique Paths II

Follow up for “Unique Paths”:

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

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

The total number of unique paths is 2.

Note: *m* and *n* will be at most 100.

Solution:
Deep search with Cache

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {

        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();

        this->f = vector<vector<int> >(m + 1, vector<int>(n + 1, 0));
        return dfs(obstacleGrid, m, n);
    }
private:
    vector<vector<int> > f;
    int dfs(const vector<vector<int> > &obstacleGrid, int x, int y) {
        if(x < 1 || y < 1) return 0;

        if (obstacleGrid[x-1][y-1]) return 0;

        if (x == 1 and y == 1) return 1;

        return getOrUpdate(obstacleGrid, x - 1, y) +
                  getOrUpdate(obstacleGrid, x, y - 1);
    }

    int getOrUpdate(const vector<vector<int> > &obstacleGrid,
              int x, int y) {
          if (f[x][y] > 0) return f[x][y];
          else return f[x][y] = dfs(obstacleGrid, x, y);
    }
};
```

Dynamic Programming
Special attention disorder first column
$O(n^2)$ runtime, O(n) space:

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {

        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();
        if (obstacleGrid[0][0] || obstacleGrid[m-1][n-1]) return 0;

        vector<int> f(n, 0);
        f[0] = obstacleGrid[0][0] ? 0 : 1;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                f[j] = obstacleGrid[i][j] ? 0 : (j == 0 ? 0 : f[j - 1]) + f[j];
            }
        }

        return f[n - 1];
    }

};
```

1.26 Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

[
[*2*],
[*3*,4],
[6,*5*,7],
[4,*1*,8,3]
]

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note:

Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

Solution:

Set state f (i, j), Represents the position from (i, j) of departure, and minimum path.

State transition equation:

f(i,j) = min{f(i,j + 1),f(i + 1,j + 1)} + (i,j)

O(n^2) runtime, O(1) space:

```
class Solution {
public:
    int minimumTotal(vector<vector<int> > &triangle) {
        for (int i = triangle.size() - 2; i >= 0; --i)
              for (int j = 0; j < i + 1; ++j)
                  triangle[i][j] += min(triangle[i + 1][j],
                          triangle[i + 1][j + 1]);
        return triangle [0][0];
    }
};
```

1.27 Subsets

Given a set of distinct integers, S, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,

If S = [1,2,3], a solution is:

[
[3],
[1],
[2],
[1,2,3],
[1,3],
[2,3],
[1,2],

[]
]

Solution:
O(2^n) runtime, O(n) space, Recursive

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {

        sort(S.begin(), S.end());
        vector<vector<int> > result;
        vector<int> path;
        subsets(S, path, 0, result);

        return result;
    }
private:
    static void subsets(const vector<int> &S, vector<int> &path, int step,
              vector<vector<int> > &result) {

                  if (step == S.size()) {
                      result.push_back(path);
                      return;
                  }

                  subsets(S, path, step + 1, result);
                  path.push_back(S[step]);
                  subsets(S, path, step + 1, result);
                  path.pop_back();
              }
};
```

O(2^n) runtime, O(n) space, Bit vector

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {

        sort(S.begin(), S.end());
        vector<vector<int> > result;
        vector<bool> selected(S.size(), false);
        subsets(S, selected, 0, result);

        return result;
    }
private:
    static void subsets(const vector<int> &S, vector<bool> &selected, int step,
              vector<vector<int> > &result) {

                  if (step == S.size()) {
                     vector<int> subset;
                     for (int i = 0; i < S.size(); i++) {
                         if (selected[i]) subset.push_back(S[i]);
                     }
                     result.push_back(subset);
                     return;
                  }

                  selected[step] = false;
                  subsets(S, selected, step + 1, result);
                  selected[step] = true;
                  subsets(S, selected, step + 1, result);
              }
};
```

O(2^n) runtime, O(1) space, Iteration

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {

        sort(S.begin(), S.end());
        vector<vector<int> > result(1);
        for (auto elem : S) {
            result.reserve(result.size() * 2);
            auto half = result.begin() + result.size();
```

```
            copy(result.begin(), half, back_inserter(result));
            for_each(half, result.end(), [&elem](decltype(result[0]) &e){
                  e.push_back(elem);
                });
        }

        return result;
    }
};
```

O(2^n) runtime, O(1) space, Binary
Elements of the collection does not exceed *int* digits. Int integer with a bit vector, the *i-th* bit is *1*, then select *S[i]*, for *0* is not selected
Bit Vector Optimization

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {

        sort(S.begin(), S.end());
        vector<vector<int> > result;
        const size_t n = S.size();
        vector<int> v;
        for (size_t i = 0; i < 1 << n; i++) {
              for (size_t j = 0; j < n; j++) {
                  if (i & 1 << j) v.push_back(S[j]);
              }
              result.push_back(v);
              v.clear();
          }

        return result;
    }
};
```

1.28 Subsets II

Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.

- The solution set must not contain duplicate subsets.

For example,

If S = [1,2,2], a solution is:

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

Solution:
In fact, it is very similar with with *Subsets*. In *Subsets*, each element can only choose either *0* or *1*, then can be expanded to up to several times to choose *0*.

$O(2^n)$ runtime, O(n) space, Recursive

```
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {

        sort(S.begin(), S.end());
        vector<vector<int> > result;
        vector<int> path;

        dfs(S, S.begin(), path, result);

        return result;
    }
private:
    static void dfs(const vector<int> &S, vector<int>::iterator start,
            vector<int> &path, vector<vector<int> > &result) {

                result.push_back(path);

                for (auto i = start; i < S.end(); i++) {
                    if (i != start && *i == *(i-1)) continue;
                    path.push_back(*i);
                    dfs(S, i + 1, path, result);
                    path.pop_back();
                }
```

```
            }
};
```

$O(2^n)$ runtime, O(n) space, Incremental Construction

```
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {

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

        unordered_map<int, int> count_map;
        for_each(S.begin(), S.end(), [&count_map](int e) {
             if (count_map.find(e) != count_map.end())
                 count_map[e]++;
             else
                 count_map[e] = 1;
         });

         vector<pair<int, int> > elems;
         for_each(count_map.begin(), count_map.end(),
                 [&elems](const pair<int, int> &e) {
                     elems.push_back(e);
         });

         sort(elems.begin(), elems.end());
         vector<int> path;

         subsets(elems, 0, path, result);
         return result;
    }
private:
    static void subsets(const vector<pair<int, int> > &elems,
            size_t step, vector<int> &path, vector<vector<int> > &result) {

                if (step == elems.size()) {
                    result.push_back(path);
                    return;
                }

                for (int i = 0; i <= elems[step].second; i++) {
```

```
                    for (int j = 0; j < i; ++j) {
                        path.push_back(elems[step].first);
                    }
                    subsets(elems, step + 1, path, result);
                    for (int j = 0; j < i; ++j) {
                        path.pop_back();
                    }
                }
            }
};
```

O(2^n) runtime, O(n) space, Bit vector

```
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {

        vector<vector<int> > result;
        sort(S.begin(), S.end());
        vector<int> count(S.back() - S.front() + 1, 0);

        for (auto i : S) {
            count[i - S[0]]++;
        }

        vector<int> selected(S.back() - S.front() + 1, -1);

        subsets(S, count, selected, 0, result);

        return result;
    }
private:
    static void subsets(const vector<int> &S, vector<int> &count,
              vector<int> &selected, size_t step, vector<vector<int> > &result) {
                  if (step == count.size()) {
                    vector<int> subset;
                    for(size_t i = 0; i < selected.size(); i++) {
                        for (int j = 0; j < selected[i]; j++) {
                            subset.push_back(i+S[0]);
                        }

                    }
```

```
                    result.push_back(subset);
                    return;

                    }

                for(int i = 0; i <= count[step]; i++) {

                    selected[step] = i;
                    subsets(S, count, selected, step + 1, result);
                }
            }
};
```

O(2^n) runtime, O(1) space, Iteration

```
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {

        sort(S.begin(), S.end());
        vector<vector<int> > result(1);

        size_t previous_size = 0;
        for (size_t i = 0; i < S.size(); ++i) {
            const size_t size = result.size();
            for (size_t j = 0; j < size; ++j) {
                if (i == 0 || S[i] != S[i-1] || j >= previous_size) {
                    result.push_back(result[j]);
                    result.back().push_back(S[i]);
                }
            }
            previous_size = size;
        }

        return result;
    }

};
```

O(2^n) runtime, O(1) space, Binary

```
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {

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

        set<vector<int> > result;
        const size_t n = S.size();
        vector<int> v;

        for (size_t i = 0; i < 1U << n; ++i) {
            for (size_t j = 0; j < n; ++j) {
                if (i & 1 << j) {
                    v.push_back(S[j]);
                }
            }
            result.insert(v);
            v.clear();
        }
        vector<vector<int> > real_result;
        copy(result.begin(), result.end(), back_inserter(real_result));
        return real_result;
    }

};
```

1.29 Spiral Matrix

Given a matrix of *m x n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

For example,

Given the following matrix:

[
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
]

You should return [1,2,3,6,9,8,7,4,5].

Solution:
O(n^2) runtime, O(1) space:

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int> > &matrix) {
        vector<int> result;
        if (matrix.empty()) return result;
        int beginX = 0, endX = matrix[0].size() - 1;
        int beginY = 0, endY = matrix.size() - 1;
        while (true) {
            // From left to right
            for (int j = beginX; j <= endX; ++j) result.push_back(matrix[beginY]\
[j]);
            if (++beginY > endY) break;
            // From top to bottom
            for (int i = beginY; i <= endY; ++i) result.push_back(matrix[i][endX\
]);
            if (beginX > --endX) break;
            // From right to left
            for (int j = endX; j >= beginX; --j) result.push_back(matrix[endY][j\
]);
            if (beginY > --endY) break;
            // From bottom to top
            for (int i = endY; i >= beginY; --i) result.push_back(matrix[i][begi\
nX]);
            if (++beginX > endX) break;
        }

        return result;
    }
};
```

1.30 Spiral Matrix II

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

For example,

Given n = 3,

You should return the following matrix:

[
[1, 2, 3],
[8, 9, 4],

[7, 6, 5]
]

Solution:
$O(n^2)$ runtime, $O(n^2)$ space:

```
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector<vector<int> > matrix(n, vector<int>(n));
        int begin = 0, end = n - 1;
        int num = 1;

        while(begin < end) {
            for (int j = begin; j < end; ++j) matrix[begin][j] = num++;
            for (int i = begin; i < end; ++i) matrix[i][end] = num++;
            for (int j = end; j > begin; --j) matrix[end][j] = num++;
            for (int i = end; i > begin; --i) matrix[i][begin] = num++;
            ++begin;
            --end;
        }

        if (begin == end) matrix[begin][begin] = num;

        return matrix;
    }
};
```

$O(n^2)$ runtime, $O(n^2)$ space:

```
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector< vector<int> > matrix(n, vector<int>(n));
        if (n == 0) return matrix;
        int beginX = 0, endX = n - 1;
        int beginY = 0, endY = n - 1;
        int num = 1;
        while (true) {
            for (int j = beginX; j <= endX; ++j) matrix[beginY][j] = num++;
            if (++beginY > endY) break;

            for (int i = beginY; i <= endY; ++i) matrix[i][endX] = num++;
```

```
            if (beginX > --endX) break;

            for (int j = endX; j >= beginX; --j) matrix[endY][j] = num++;
            if (beginY > --endY) break;

            for (int i = endY; i >= beginY; --i) matrix[i][beginX] = num++;
            if (++beginX > endX) break;
        }
        return matrix;
    }
};
```

1.31 Sort Colors

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:

You are not suppose to use the library's sort function for this problem.

Solution:
Counting Sort O(n) runtime, O(1) space:

```
class Solution {
public:
    void sortColors(int A[], int n) {
        int counts[3] = { 0 };

        for (int i = 0; i < n; i++) {
            counts[A[i]]++;
        }

        for (int i = 0, index = 0; i < 3; i++) {
            for (int j = 0; j < counts[i]; j++) {
                A[index++] = i;
            }
        }
    }
};
```

Double pointer
O(n) runtime, O(1) space:

```
class Solution {
public:
    void sortColors(int A[], int n) {
        int red = 0, blue = n - 1;

        for (int i = 0; i < blue + 1;) {

            if (A[i] == 0) {
                swap(A[i++], A[red++]);
            }else if(A[i] == 2) {
                swap(A[i], A[blue--]);
            }else {
                i++;
            }
        }
    }
};
```

use partition()
O(n) runtime, O(1) space:

```
class Solution {
public:
    void sortColors(int A[], int n) {
        partition(partition(A, A + n, bind1st(equal_to<int>(), 0)), A + n,
                    bind1st(equal_to<int>(), 1));
    }
};
```

Reimplement partition()
O(n) runtime, O(1) space:

```
class Solution {
public:
    void sortColors(int A[], int n) {
        partition(partition(A, A + n, bind1st(equal_to<int>(), 0)), A + n,
                    bind1st(equal_to<int>(), 1));
    }
private:
    template<typename ForwardIterator, typename UnaryPredicate>
    ForwardIterator partition(ForwardIterator first, ForwardIterator last,
              UnaryPredicate pred) {
```

```
11
12                  auto pos = first;
13                  for (; first != last; ++first) {
14                      if (pred(*first)) {
15                          swap(*first, *pos++);
16                      }
17                  }
18
19                  return pos;
20              }
21  };
```

1.32 Pascal's Triangle

Given *numRows*, generate the first numRows of Pascal's triangle.

For example, given *numRows* = 5,

Return
[
[1]
[1,1],
[1,2,1],
[1,3,3,1],
[1,4,6,4,1]
]

Solution:

From left to right
$O(n^2)$ runtime, O(n) space:

```
1   class Solution {
2   public:
3       vector<vector<int> > generate(int numRows) {
4           vector<vector<int> > result;
5
6           if(numRows == 0) return result;
7
8           result.push_back(vector<int>(1,1));
9
10          for(int i = 2; i <= numRows; ++i) {
11              vector<int> current(i,1);
12              const vector<int> &prev = result[i-2];
```

```
13
14              for(int j = 1; j < i - 1; ++j) {
15                  current[j] = prev[j-1] + prev[j];
16              }
17              result.push_back(current);
18          }
19          return result;
20      }
21  };
```

From right to left
O(n^2) runtime, O(n) space:

```
1   class Solution {
2   public:
3       vector<vector<int> > generate(int numRows) {
4           vector<vector<int> > result;
5             vector<int> array;
6             for (int i = 1; i <= numRows; i++) {
7                 for (int j = i - 2; j > 0; j--) {
8                     array[j] = array[j - 1] + array[j];
9                 }
10                array.push_back(1);
11                result.push_back(array);
12            }
13            return result;
14      }
15  };
```

1.33 Pascal's Triangle II

Given an index *k*, return the k^{th} row of the Pascal's triangle.

For example, given k = 3,

Return [1,3,3,1].

NOTE:

Could you optimize your algorithm to use only O(k) extra space?

Solution:

Scroll Array
O(n^2) runtime, O(n) space:

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> array;
        for (int i = 0; i <= rowIndex; i++) {
            for (int j = i - 1; j > 0; j--){
                array[j] = array[j - 1] + array[j];
            }
            array.push_back(1);
        }
        return array;
    }
};
```

1.34 Search Insert Position

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

Solution:

Reimplement lower_bound
O(logn) runtime, O(1) space:

```
class Solution {
public:
    int searchInsert(int A[], int n, int target) {
        return lower_bound(A, A + n, target) - A;
    }
private:
    template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound (ForwardIterator first,
              ForwardIterator last, T value) {

```

```
                while (first != last) {
                    auto mid = next(first, distance(first, last) / 2);

                    if (value > *mid) {
                        first = ++mid;
                    }else {
                        last = mid;
                    }
                }
                return first;
            }
};
```

1.35 Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of *O(log n)*.

If the target is not found in the array, return [-1, -1].

For example,

Given [5, 7, 7, 8, 8, 10] and target value 8,

return [3, 4].

Solution:

STL
O(logn) runtime, O(1) space:

```
class Solution {
public:
    vector<int> searchRange(int A[], int n, int target) {
        const int l = distance(A, lower_bound(A, A + n, target));
        const int u = distance(A, prev(upper_bound(A, A + n, target)));

        if (A[l] != target) {
            return vector<int> { -1, -1 };
        }else {
            return vector<int> { l, u };
        }
    }
};
```

1.36 Search a 2D Matrix

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

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

For example,

Consider the following matrix:

[
[1, 3, 5, 7],
[10, 11, 16, 20],
[23, 30, 34, 50]
]

Given target = 3, return true.

Solution:

Binary Search
O(logn) runtime, O(1) space:

```
1  class Solution {
2  public:
3      bool searchMatrix(vector<vector<int> > &matrix, int target) {
4          if (matrix.empty()) return false;
5
6          const size_t  m = matrix.size();
7          const size_t n = matrix.front().size();
8
9          int first = 0;
10         int last = m * n;
11
12         while (first < last) {
13             int mid = first + (last - first) / 2;
14             int value = matrix[mid / n][mid % n];
15
16             if (value == target) {
17                 return true;
18             }else if(value < target) {
19                 first = mid + 1;
```

```
            }else {
                last = mid;
            }
        }

        return false;
    }
};
```

1.37 Minimum Path Sum

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note:

You can only move either down or right at any point in time.

Solution:

Deep Search with Cache

```
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        const int m = grid.size();
        const int n = grid[0].size();

        this->f = vector<vector<int> >(m, vector<int>(n, -1));
        return dfs(grid, m-1, n-1);
    }
private:
    vector<vector<int> > f;

    int dfs(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX;

        if (x == 0 && y == 0) return grid[0][0];

        return min(getOrUpdate(grid, x - 1, y),
                   getOrUpdate(grid, x, y - 1)) + grid[x][y];
    }

    int getOrUpdate(const vector<vector<int> > &grid, int x, int y)  {
```

```
        if (x < 0 || y < 0) return INT_MAX;

        if (f[x][y] >= 0) return f[x][y];
        else return f[x][y] = dfs(grid, x, y);
    }
};
```

Dynamic Programming

```
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        if (grid.size() == 0) return 0;

        const int m = grid.size();
        const int n = grid[0].size();

        int f[m][n];
        f[0][0] = grid[0][0];

        for (int i = 1; i < m; i++) {
            f[i][0] = f[i - 1][0] + grid[i][0];
        }

        for(int i = 1; i < n; i++) {
            f[0][i] = f[0][i - 1] + grid[0][i];
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[i][j] = min(f[i - 1][j], f[i][j - 1]) + grid[i][j];
            }
        }

        return f[m - 1][n - 1];
    }

};
```

Deep Search with Cache + Dynamic Programming

```
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
       const int m = grid.size();
       const int n = grid[0].size();

       int f[n];
       fill(f, f+n, INT_MAX);
       f[0] = 0;

       for (int i = 0; i < m; i++) {
           f[0] += grid[i][0];
           for (int j = 1; j < n; j++) {
               f[j] = min(f[j - 1], f[j]) + grid[i][j];
           }
       }

       return f[n - 1];
    }

};
```

1.38 Maximum Subarray

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

For example, given the array [−2,1,−3,4,−1,2,1,−5,4],

the contiguous subarray [4,−1,2,1] has the largest sum = 6.

Solution:

Set status is *f[j]*, represented by *S[j]* at the end of the maximum continuous sequence and then the state transition equation is as follows:

f[j] = max{ f[j−1]+S[j],S[j] },1≤j≤n

target = max{f[j]},1≤j≤n

Dynamic Programming
O(n) runtime, O(1) space:

```
class Solution {
public:
    int maxSubArray(int A[], int n) {
        int result = INT_MIN, f = 0;
          for (int i = 0; i < n; ++i) {
              f = max(f + A[i], A[i]);
              result = max(result, f);
          }
          return result;
    }
};
```

O(n) runtime, O(n) space:

```
class Solution {
public:
    int maxSubArray(int A[], int n) {

        return mcss(A, n);
    }
private:
    static int mcss(int A[], int n)  {

        int i, result, cur_min;
        int *sum=new int[n+1];

        sum[0] = 0;
        result = INT_MIN;
        cur_min = sum[0];
        for (i = 1; i <= n; i++) {
            sum[i] = sum[i - 1] + A[i - 1];
        }
        for (i = 1; i <= n; i++) {
            result = max(result, sum[i] - cur_min);
            cur_min = min(cur_min, sum[i]);
        }
        delete[] sum;
        return result;
    }
};
```

1.39 Maximum Product Subarray

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.

Solution:

The idea is similar with "Find the subarray wich has the largest sum

The only thing to note here is, maximum product can also be obtained by minimum (negative) product ending with the previous element multiplied by this element. For example, in array {12, 2, -3, -5, -6, -2}, when we are at element -2, the maximum product is multiplication of, minimum product ending with -6 and -2.

```
class Solution {
public:
    int maxProduct(int A[], int n) {
        // To remember the max/min product for previous position
        int maxPrev = A[0], minPrev = A[0];
        // To remember the max/min product for current position
        int maxHere = A[0], minHere = A[0];
        // Overall maximum product
        int maxProd = A[0];

        for (int i=1; i<n; i++){
            //max( maxPrev * A[i],  minPrev * A[i],  A[i] )
            maxHere = max( max( maxPrev * A[i], minPrev * A[i] ), A[i] );
            //min( maxPrev * A[i],  minPrev * A[i],  A[i] )
            minHere = min( min( maxPrev * A[i], minPrev * A[i] ), A[i] );
            //Keep tracking the overall maximum product
            maxProd = max(maxHere, maxProd);
            //Shift the current max/min product to previous variables
            maxPrev = maxHere;
            minPrev = minHere;
        }
        return maxProd;
    }
};
```

1.40 Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

Solution:

O(n^2) runtime, O(n) space:

```
class Solution {
public:
    int maximalRectangle(vector<vector<char> > &matrix) {
        if (matrix.empty())  return 0;

        const int m = matrix.size();
        const int n = matrix[0].size();
        vector<int> H(n, 0);
        vector<int> L(n, 0);
        vector<int> R(n, n);

        int ret = 0;
        for(int i = 0; i < m; ++i) {
            int left = 0, right = n;
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == '1') {
                    ++H[j];
                    L[j] = max(L[j], left);
                }else{
                    left = j+1;
                    H[j] = 0; L[j] = 0; R[j] = n;
                }
            }

            for (int j = n-1; j >= 0; --j) {
                if (matrix[i][j] == '1') {
                    R[j] = min(R[j], right);
                    ret = max(ret, H[j]*(R[j]-L[j]));
                }else {
                    right = j;
                }
            }
        }

```

```
35          return ret;
36      }
37  };
```

1.41 Majority Element

Given an array of size *n*, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.

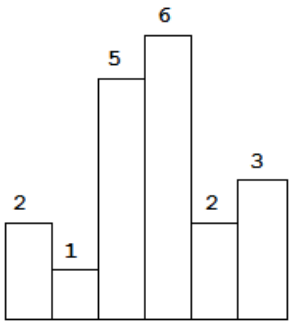
You may assume that the array is non-empty and the majority element always exist in the array.

Solution:

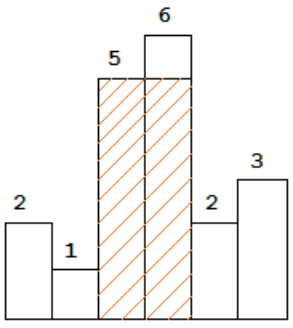
<<(code/MajorityElement.cpp)

1.42 Largest Rectangle in Histogram

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.



The largest rectangle is shown in the shaded area, which has area = `10` unit.

For example,

Given height = [2,1,5,6,2,3],

return 10.

Solution:

Current element is greater than the topelement, the push, otherwirse merge existing stack until the top element is less than the current element. When the end of the stack element 0, repeat once the merger.

O(n) runtime, O(n) space:

```
class Solution {
public:
    int largestRectangleArea(vector<int> &height) {
        stack<int> s;
        height.push_back(0);
        int result = 0;
        for (int i = 0; i < height.size(); ) {
            if (s.empty() || height[i] > height[s.top()]) {
                s.push(i++);
            }else {
                int tmp = s.top();
                s.pop();
                result = max(result,
                             height[tmp] * (s.empty() ? i : i - s.top() - 1));
            }
        }

        return result;
    }
};
```

1.43 Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:

A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.

Solution:

Greedy Algorithm
O(n) runtime, O(1) space:

```
class Solution {
public:
    bool canJump(int A[], int n) {
        int reach = 1;
        for (int i = 0; i < reach && reach < n; ++i) {
            reach = max(reach,  i + 1 + A[i]);
        }

        return reach >= n;
    }
};
```

O(n) runtime, O(1) space:

```
class Solution {
public:
    bool canJump(int A[], int n) {
        if (n == 0) return true;

        int left_most = n - 1;

        for (int i = n - 2; i >= 0; --i) {
            if (i + A[i] >= left_most) {
                left_most = i;
            }
        }

        return left_most == 0;
    }
};
```

Dynamic Programming
O(n) runtime, O(n) space:

```
class Solution {
public:
    bool canJump(int A[], int n) {
        vector<int> f(n, 0);
        f[0] = 0;
        for (int i = 1; i < n; i++) {
            f[i] = max(f[i - 1], A[i - 1]) - 1;
            if (f[i] < 0) return false;
        }

        return f[n - 1] >= 0;
    }
};
```

1.44 Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:

Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

Solution:
Greedy Algorithm
O(n) runtime, O(1) space:

```
class Solution {
public:
    int jump(int A[], int n) {
        int step = 0;
        int left = 0;
        int right = 0;
        if (n == 1) return 0;

        while (left <= right) {
            ++step;
            const int old_right = right;
            for (int i = left; i <= old_right; ++i) {
```

```
                int new_right = i + A[i];
                if (new_right >= n - 1) return step;

                if (new_right > right) right = new_right;
            }
            left = old_right + 1;
        }
        return 0;
    }
};
```

O(n) runtime, O(1) space:

```
class Solution {
public:
    int jump(int A[], int n) {
        int result = 0;
        // the maximum distance that has been reached
        int last = 0;
        // the maximum distance that can be reached by using "ret+1" steps
        int cur = 0;

        for (int i = 0; i < n; ++i) {
            if (i > last) {
                last = cur;
                ++result;
            }
            cur = max(cur, i + A[i]);
        }
        return result;
    }
};
```

1.45 Insert Interval

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1:

Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2:

Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

Solution:

O(n) runtime, O(1) space:
That's maybe timeout

```
/**
 * Definition for an interval.
 * struct Interval {
 *     int start;
 *     int end;
 *     Interval() : start(0), end(0) {}
 *     Interval(int s, int e) : start(s), end(e) {}
 * };
 */
class Solution {
public:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {

        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {

            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            }else if(newInterval.start > it->end) {
                it++;
                continue;
            }else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

Merge

```
/**
 * Definition for an interval.
 * struct Interval {
 *     int start;
 *     int end;
 *     Interval() : start(0), end(0) {}
 *     Interval(int s, int e) : start(s), end(e) {}
 * };
 */

bool compare(const Interval& lhs, const Interval& rhs){
    return (lhs.start==rhs.start) ? lhs.end < rhs.end : lhs.start < rhs.start;
}

vector<Interval> merge(vector<Interval> &intervals) {

    vector<Interval> result;

    if (intervals.size() <= 0) return result;
    //sort the inervals. Note: using the customized comparing function.
    sort(intervals.begin(), intervals.end(), compare);
    for(int i=0; i<intervals.size(); i++) {
        int size = result.size();
        // if the current intervals[i] is overlapped with previous interval.
        // merge them together
        if( size>0 && result[size-1].end >= intervals[i].start) {
            result[size-1].end = max(result[size-1].end, intervals[i].end);
        }else{
            result.push_back(intervals[i]);
        }
    }

    return result;
}


class Solution {
public:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {

        intervals.push_back(newInterval);

```

```
43          return merge(intervals);
44      }
45  };
```

1.46 First Missing Positive

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

For example,

Given [1,2,0] return 3,

and [3,4,-1,1] return 2.

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

Solution:

Bucket Sort
When *A [i]! = I + 1*, the *A [i]* and *A [A [i] -1]* exchange, the exchange can not be so far until the termination condition is *A [i] == A [A [i] -1]*.

O(n) runtime, O(1) space:

```
1   class Solution {
2   public:
3       int firstMissingPositive(int A[], int n) {
4           bucket_sort(A, n);
5
6           for (int i = 0; i < n; ++i) {
7               if (A[i] != (i + 1)) {
8                   return i + 1;
9               }
10          }
11          return n + 1;
12      }
13  private:
14      static void bucket_sort(int A[], int n) {
15          for (int i = 0; i < n; i++) {
16              while (A[i] != i + 1) {
17                  if (A[i] <= 0 || A[i] > n || A[i] == A[A[i] - 1]) {
18                      break;
19                  }
20                  swap(A[i], A[A[i] - 1]);
21              }
22          }
```

```
    }
};
```

1.47 Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array where num[i] ≠ num[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

Note:

Your solution should be in logarithmic complexity.

Solution:

Binary search is common idea here.

However, you need to think about two senarios:

- Becasue we need check `num[mid-1]`, `num[mid]`, `num[mid+1]`, So, we need make sure there hasn't out-of-boundary issue.
- There are multiple Peak elements. For example: [1,2,1,2,1], or [1,2,3,1,2,1], LeetCode doesn't tell you what the expected result is. I guess:for [1,2,1,2,1] you can return either 1 or 3, because both them are peak elements, for [1,2,3,2,4,2,1] it should return 4, because num[4] is the real peak. but Leetcode accept either 2 or 4.

```
class Solution {
public:
    int findPeakElement(const vector<int> &num) {
        int n = num.size();
        int low = 0;
        int high = n - 1;

        int mid = 0, v1, v2;

        while ( low < high ) {

            // Find the index of middle element
            mid = low + ( high - low ) / 2;
```

```
14
15              // Compare middle element with its neighbours (if neighbours exist)
16              if ( ( mid == 0 || num[mid] > num[mid-1] ) &&
17                  ( mid == n-1 ||  num[mid] > num[mid+1] )  ){
18                  return mid;
19              }
20
21              // If middle element is not peak and its left neighbor is greater th\
22  an it
23              // then left half must have a peak element
24              if (mid >0 && num[mid-1] > num[mid]){
25                  high = mid - 1;
26              // If middle element is not peak and its right neighbor is greater t\
27  han it
28              // then right half must have a peak element
29              }else{
30                  low = mid + 1;
31              }
32
33          }
34
35          return low;
36      }
37  };
```

1.48 Find Minimum in Rotated Sorted Array

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

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

Find the minimum element.

You may assume no duplicate exists in the array.

Solution:

Obveriously, to search any sorted array, the binary search is the common sense.

```
class Solution {
public:
    int findMin(vector<int> &num) {
        int low=0, high=num.size()-1;

        while(high-low>1){
            int mid = low + (high-low)/2;
            // Chek the array if it is non-rotated, then just return the first e\
lement.
            if (num[low] < num[mid] && num[mid] < num[high]){
                return num[low];
            }

            // The array is rotated
            // Spli it into two part, the minimal value must be the rotated part

            // if the left part is rotated, warch the left part
            if (num[low] > num [mid]){
                high = mid;
                continue;
            }
            // if the right part is rotated, search the right part.
            if (num[mid] > num[high]){
                low = mid;
                continue;
            }
        }
        // the array only has 1 element
        if (high == low) return num[low];

        // the array has 2 elements
        return num[low] < num[high] ? num[low] : num[high];
    }
};
```

1.49 Find Minimum in Rotated Sorted Array II

Follow up for “Find Minimum in Rotated Sorted Array”:

What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

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.

Solution:

<<(code/FindMinimuminRotatedSortedArrayII.cpp)

1.50 Container With Most Water

Given *n* non-negative integers *a1, a2, ..., an*, where each represents a point at coordinate *(i, ai)*. *n* vertical lines are drawn such that the two endpoints of line i is at *(i, ai)* and *(i, 0)*.

Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note:

You may not slant the container.

Solution:

O(n) runtime, O(1) space:

```
class Solution {
public:
    int maxArea(vector<int> &height) {
          int start = 0;
          int end = height.size() - 1;
          int result = INT_MIN;
          while (start < end) {
              int area = min(height[end], height[start]) * (end - start);
              result = max(result, area);
              if (height[start] <= height[end]) {
                  start++;
              } else {
                  end--;

              }
          }
          return result;
    }
};
```

1.51 Construct Binary Tree from Preorder and Inorder Traversal

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

Note:

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

Solution:

O(n) runtime, O(logn) space:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {

        return buildTree(begin(preorder), end(preorder),
                  begin(inorder), end(inorder));
    }

    template<typename InputIterator>
      TreeNode* buildTree(InputIterator pre_first, InputIterator pre_last,
              InputIterator in_first, InputIterator in_last) {
          if (pre_first == pre_last) return nullptr;
          if (in_first == in_last) return nullptr;
          auto root = new TreeNode(*pre_first);
          auto inRootPos = find(in_first, in_last, *pre_first);
          auto leftSize = distance(in_first, inRootPos);
          root->left = buildTree(next(pre_first), next(pre_first,
                  leftSize + 1), in_first, next(in_first, leftSize));
          root->right = buildTree(next(pre_first, leftSize + 1), pre_last,
                  next(inRootPos), in_last);
          return root;
      }
};
```

1.52 Construct Binary Tree from Inorder and Postorder Traversal

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

Note:

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

Solution:

O(n) runtime, O(logn) space:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
        return buildTree(begin(inorder), end(inorder),
                  begin(postorder), end(postorder));
    }

    template<typename BidiIt>
      TreeNode* buildTree(BidiIt in_first, BidiIt in_last,
              BidiIt post_first, BidiIt post_last) {
          if (in_first ==in_last) return nullptr;
          if (post_first == post_last) return nullptr;
          const auto val = *prev(post_last);
          TreeNode* root = new TreeNode(val);
          auto in_root_pos = find(in_first, in_last, val);
          auto left_size = distance(in_first, in_root_pos);
          auto post_left_last = next(post_first, left_size);
          root->left = buildTree(in_first, in_root_pos, post_first, post_left_la\
st);
          root->right = buildTree(next(in_root_pos), in_last, post_left_last,
                  prev(post_last));
          return root;
```

```
32          }
33  };
```

1.53 Combination Sum

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

The **same** repeated number may be chosen from **C** unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination *(a1, a2, … , ak)* must be in non-descending order. *(ie, a1 ≤ a2 ≤ … ≤ ak)*.
- 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]`

Solution:

<<(code/CombinationSum.cpp)

1.54 Combination Sum II

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

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

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination *(a1, a2, … , ak)* must be in non-descending order. *(ie, a1 ≤ a2 ≤ … ≤ ak)*.
- 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]
```

Solution:

```
void combinationSumHelper(vector<int> &candidates, int start, int target, vector\
<int> &solution, vector< vector<int> > &result) {
    if (target<0){
        return;
    }
    if (target==0){
        result.push_back(solution);
    }
    for(int i=start; i<candidates.size(); i++){
        //skip duplicates
        int n = candidates[i];
        if (i>start && candidates[i] == candidates[i-1]) {
            continue;
        }
        solution.push_back(n);
        combinationSumHelper(candidates, i+1, target - n, solution, result);
        solution.pop_back();
    }
}

class Solution {
public:
    vector<vector<int> > combinationSum2(vector<int> &num, int target) {
        vector< vector<int> > result;
        if (num.size()<=0){
            return result;
        }
        sort(num.begin(), num.end());

        vector<int> solution;
        combinationSumHelper(num, 0, target, solution, result);

```

```
        return result;
    }
};
```

1.55 Best Time to Buy and Sell Stock

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.

Solutuon:

Greedy Algorithm
O(n) runtime, O(1) space:

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if (prices.size() < 2) return 0;
        int profit = 0;
        int cur_min = prices[0];

        for (int i = 1; i < prices.size(); i++) {
            profit = max(profit, prices[i] - cur_min);
            cur_min = min(cur_min, prices[i]);
        }
        return profit;
    }
};
```

1.56 Best Time to Buy and Sell Stock II

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution:

Greedy Algorithm
O(n) runtime, O(1) space:

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int sum = 0;
        for (int i = 1; i < prices.size(); i++) {
              int diff = prices[i] - prices[i - 1];
              if (diff > 0) sum += diff;
        }
        return sum;
    }
};
```

1.57 Best Time to Buy and Sell Stock III

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution:

max{ f(i) + g(i) },0 ≤ i ≤ n − 1

O(n) runtime. O(n) space:

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if (prices.size() < 2) return 0;

        const int n = prices.size();
        vector<int> f(n, 0);
        vector<int> g(n, 0);

        for (int i = 1, valley = prices[0]; i < n; ++i) {
            valley = min(valley, prices[i]);
            f[i] = max(f[i - 1], prices[i] - valley);
        }

        for (int i = n - 2, peak = prices[n - 1]; i >= 0; --i) {
            peak = max(peak, prices[i]);
```

```
            g[i] = max(g[i], peak - prices[i]);
        }

        int max_profit = 0;
        for (int i = 0; i < n; ++i) {
            max_profit = max(max_profit, f[i] + g[i]);
        }

        return max_profit;
    }
};
```