

Haoda Fu  
Eli Lilly and Company

# Algorithms: Problems and Solutions

Implemented in C++11

July 22, 2018

Springer



*This book is dedicated to my wife Jie Cui,  
and two children Sophia Fu and Valerie Fu.*



# Preface

- who should read this book.
- why I am writing this book. Story starts from my new job responsibility. Need elegant solution.
- why in C++11

I am very grateful to those who have reviewed the text or who have sent me constructive suggestions and corrections.

Carmel, Indiana,

*Haoda Fu*  
Oct. 2017



# Contents

<b>1</b>	<b>Linear List</b>	1
1.1	Array	1
1.1.1	Two Sum	1
1.1.2	3Sum	3
1.1.3	3Sum Closest	4
1.1.4	4Sum	5
1.1.5	Remove Element	7
1.1.6	Remove Duplicates from Sorted Array	8
1.1.7	Remove Duplicates from Sorted Array II	9
1.1.8	Search in Rotated Sorted Array	11
1.1.9	Search in Rotated Sorted Array II	12
1.1.10	Median of Two Sorted Arrays	13
1.1.11	Longest Consecutive Sequence	14
1.1.12	Container With Most Water	16
1.1.13	Search Insert Position	17
1.1.14	Plus One	18
1.1.15	Climbing Stairs	19
1.1.16	Merge Two Sorted Arrays	21
1.2	Singly Linked List	22
1.2.1	Add Two Numbers	22
1.2.2	Merge Two Sorted Arrays	23
1.2.3	Remove Duplicates from Sorted List	24
<b>2</b>	<b>String</b>	27
2.1	Longest Substring Without Repeating Characters	27
2.2	Palindrome Number	28
2.3	Longest Palindromic Substring	29
2.4	ZigZag Conversion	32
2.5	String to Integer (atoi)	34
2.6	Regular Expression Matching	35
2.7	Integer to Roman	36

2.8	Roman to Integer .....	37
2.9	Longest Common Prefix .....	37
2.10	Implement strStr() .....	39
2.11	Count and Say .....	40
2.12	Maximum Subarray .....	41
2.13	Length of Last Word .....	43
2.14	Add Binary .....	43
<b>3</b>	<b>Tree</b> .....	45
3.1	Binary Tree Traversals .....	45
3.1.1	Same Tree .....	45
3.1.2	Symmetric Tree .....	47
3.2	Construction of Binary Tree .....	48
3.3	Binary Search Tree .....	48
3.4	Recursion with a Binary Tree .....	48
3.4.1	Maximum Depth of Binary Tree .....	48
<b>4</b>	<b>Stack and Queue</b> .....	51
4.1	Stack .....	51
4.1.1	Valid Parentheses .....	51
4.2	Queue .....	52
<b>5</b>	<b>Details Implementation</b> .....	53
5.1	Reverse Integer .....	53
5.2	Sqrt(x) .....	54
<b>A</b>	<b>Chapter Heading</b> .....	57
A.1	Section Heading .....	57
A.1.1	Subsection Heading .....	57
	<b>Glossary</b> .....	59



# Chapter 1

## Linear List

### 1.1 Array

#### *1.1.1 Two Sum*

##### **Description**

(1 easy) Given an array of integers, return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution, and you may not use the same element twice.

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

##### **Analysis**

One naive implementation is to use a brutal force with two loops to check the combination. Its time complexity is  $O(n^2)$  and space complexity is  $O(1)$ . To achieve faster speed, we can use a hash table to store the values. There might be duplicate values in the original array, we may consider to use `unordered_multimap`. Since each input has exactly one solution, we can further simplify the problem using `unordered_map` to remove the duplicates (think about why). The search for an `unordered_map` or `multimap` is  $O(1)$ . Therefore, we can trade space complexity for time complexity.

##### **Code 1: naive implementation**

```
class Solution {
public:
```

```

vector<int> twoSum(vector<int>& nums, int target)
{
    vector<int> index(2);
    for(auto iter1=nums.begin(); iter1!=nums.
end(); ++iter1){
        for(auto iter2=iter1+1; iter2!=nums.
end(); ++iter2){
            if(*iter1+*iter2==target){
                index[0]=(int) (iter1-nums
.begin());
                index[1]=(int) (iter2-nums
.begin());
                return index;
            }
        }
    }
};

```

This solution has time complexity  $O(n^2)$  and space complexity  $O(1)$ .

#### Code 2\*: using hash table

```

class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target)
    {
        unordered_map<int, int> hash;
        vector<int> result(2);
        for(int iter=0; iter!=nums.size(); ++iter){
            hash[nums[iter]]=iter;
        }
        for(int iter=0; iter!=nums.size(); ++iter){
            int gap=target-nums[iter];
            if(hash.find(gap)!=hash.end() && hash
[gap] !=iter){
                result[0]=iter;
                result[1]=hash[gap];
                break;
            }
        }
        return result;
    }
};

```

This solution has time complexity  $O(n)$  and space complexity  $O(n)$ . Please pay special attention on why we can use `unordered_map` which only store unique keys.

### 1.1.2 3Sum

#### Description

(15 medium) 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: The solution set must not contain duplicate triplets.

Example: For example, given array  $S = [-1, 0, 1, 2, -1, -4]$ , a solution set is:

```
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

#### Analysis

A simple solution is to solve this using similar idea as the in section 1.1.1. We can set the first value as fixed target, and search for the second and third value. We noticed that this problem does not require the indices, so that we can sort the array first, and use contraction methods to find solutions with time complexity  $O(n^2)$ . In addition, the summation of the 3 numbers are required to be zero which implies that the largest number has to be greater or equal to zero and the smallest number should be less or equal to zero.

#### Code

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

```

        while(*k==*(k+1) && j<k && *k>=0)
            --k;
    } else {
        result.push_back({*i,*j,*k});
        ++j;
        --k;
        while(*j==*(j-1) && *k==*(k+1) && j
            <k && *k>=0) {
            ++j;
            --k;
        }
    }
}
return result;
}
};

```

### 1.1.3 3Sum Closest

#### Description

(16 medium) 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.

Example: given array  $S = \{-1, 2, 1, -4\}$ , and target=1. The sum that is closest to the target is 2 ( $-1 + 2 + 1 = 2$ ).

#### Analysis

For a sorted array, we can use contraction method to find closest target. The time complexity is  $O(n^2)$ . We can move the first pointer from the smallest value toward larger values. Conditional on the first pointer, we apply contraction methods to search a sum of 3 to the closest value of the target.

#### Code

```

class Solution {
public:
    int threeSumClosest(vector<int>& nums, int target) {
        int min_dist=INT_MAX;
        int result = INT_MAX;
        if (nums.size() < 3) return INT_MIN;
        if (nums.size() == 3) return accumulate(nums.begin(),
            nums.end(), 0);
    }
};

```

```

sort(nums.begin(), nums.end());
for (auto i = nums.begin(); i != nums.end() - 2; ++i) {
    auto j = i + 1;
    auto k = nums.end() - 1;
    while (j < k) {
        int sum = *i + *j + *k;
        int dist = abs(sum - target);
        if (dist < min_dist) {
            min_dist = dist;
            result = sum;
        }
        if (sum > target) {
            --k;
        } else if (sum < target) {
            ++j;
        } else {
            return sum;
        }
    }
}
return result;
}
};

```

### 1.1.4 4Sum

#### Description

(18 medium) Given an array  $S$  of  $n$  integers, are there elements  $a, b, c$ , and  $d$  in  $S$  such that  $a + b + c + d = \text{target}$ ? Find all unique quadruplets in the array which gives the sum of target. Note: The solution set must not contain duplicate quadruplets.

Example: For example, given array  $S = [1, 0, -1, 0, -2, 2]$ , and  $\text{target} = 0$ . A solution set is:

```

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

```

#### Analysis

One idea is to sort the array first and add one more loop to the 3Sum (1.1.2) problem. This solution has time complexity as  $O(n^3)$ . Another approach is to use `unordered_multimap` to save the sum of two numbers. Since we have 4 numbers to reach the target, we can use the same hash table. This solution has time complexity as  $O(n^2)$ .

**Code: Contraction method**

```

class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target)
    {
        vector<vector<int>> result;
        if(nums.size()<4) return result;
        sort(nums.begin(),nums.end());
        for(auto i=nums.begin();i<nums.end()-3;++i){
            for(auto j=i+1;j<nums.end()-2;++j){
                auto k=j+1;
                auto l=nums.end()-1;
                while(k<l){
                    int sum= *i+*j+*k+*l;
                    if(sum > target){
                        --l;
                    } else if(sum<target){
                        ++k;
                    } else {
                        result.push_back({*i,*j,*k,*l});
                        ++k;
                        --l;
                    }
                }
            }
        }
        sort(result.begin(),result.end());
        result.erase(unique(result.begin(),result.end()),
            result.end());
        return result;
    }
};

```

We also should notice that we should sort the `result` first before using `unique` function because in `unique` removes all but the first element from every *consecutive* group of equivalent elements in the range. The removal is done by replacing the duplicate elements by the next element that is not a duplicate, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element.

**Code\*: Using a Hash table**

```

class Solution {
public:
    vector<vector<int>> fourSum(vector<int> &nums, int target)
    {
        vector<vector<int>> result;
        if(nums.size()<4) return result;
        unordered_multimap<int, pair<int, int>> hash;
        for(int i=0;i<nums.size()-1;++i){

```

```

        for(int j=i+1;j< nums.size();++j){
            hash.insert(make_pair(nums[i]+nums[j],
                                   make_pair(i,j)));
        }
    }
    for(auto i:hash){
        int j= target- i.first;
        auto sets = hash.equal_range(j);
        for(auto x=sets.first;x!=sets.second;++x){
            if(i.second.first!= (*x).second.first &&
               i.second.first!= (*x).second.second && i.
               second.second!= (*x).second.first && i.
               second.second!= (*x).second.second){
                vector<int> temp({nums[i.second.
                                first],nums[i.second.second],nums
                                [(*x).second.first], nums[(*x).
                                second.second]});
                sort(temp.begin(),temp.end());
                result.push_back(temp);
            }
        }
    }
    sort(result.begin(),result.end());
    result.erase(unique(result.begin(),result.end()),
                 result.end());
    return result;
}
};

```

### 1.1.5 Remove Element

#### Description

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

Example: Given input array `nums=[3,2,2,3]`, `val=3`.

Your function should return `length = 2`, with the first two elements of `nums` being 2.

#### Analysis

We can directly use STL `remove` function which transforms the range `[first,last)` into a range with all the elements that compare equal to target value removed, and returns an iterator to the new end of that range.

**Code\*: Using STL**

```

class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        return (int)distance(nums.begin(), remove(nums.begin(),
            , nums.end(), val));
    }
};

```

**Code: Direct implementation**

```

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

```

***1.1.6 Remove Duplicates from Sorted Array*****Description**

(26 easy) Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory. For example, Given input array nums = [1, 1, 2], Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

**Code 1\*: Using STL**

```

class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        return (int)distance(nums.begin(), unique(nums.begin(),
            , nums.end()));
    }
};

```



Here, `unique` is used to eliminate all but the first element from every consecutive group of equivalent elements from the range `[first, last)` and returns a past-the-end iterator for the new logical end of the range. Removing is done by shifting the elements in the range in such a way that elements to be erased are overwritten. Relative order of the elements that remain is preserved and the physical size of the container is unchanged. Iterators pointing to an element between the new logical end and the physical end of the range are still dereferenceable, but the elements themselves have unspecified values. A call to `unique` could be followed by a call to a container's `erase` method, which erases the unspecified values and reduces the physical size of the container to match its new logical size.

## Code 2: Direct Implementation

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int length = 0;
        if(nums.empty()) return 0;
        for(unsigned long iter=0; iter<nums.size(); ++iter){
            if (nums[length]!=nums[iter]) nums[++length]=
                nums[iter];
        }
        return ++length;
    }
};
```

Time complexity is  $O(n)$ , and space complexity is  $O(1)$ .

### 1.1.7 Remove Duplicates from Sorted Array II

#### Description

(80 medium) Follow up for “Remove Duplicates”: What if duplicates are allowed at most twice? For example, Given sorted array `nums = [1, 1, 1, 2, 2, 3]`, Your function should return `length = 5`, with the first five elements of `nums` being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.

#### Analysis

For these problem, we need pay special attention to the beginning and to the end. For example, what if the input is empty, or `nums = [1]` or `nums = [1, 1]`. To avoid the complexity of handling it within the iteration, if the `nums.size() <= 2`, we can directly output the length. We need to iterate through the vector, and there are two approaches. One is to focus on conditions to move forward the index (include implementation) and the other way is to find its complement condition that not moving forward the index (preclude implementation). We implement these approaches below

**Code 1: include implementation**

Time complexity is  $O(n)$ , and space complexity is  $O(1)$ .

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.size() < 2) return (int)nums.size();
        int index=1;
        for(unsigned iter=2; iter<nums.size(); ++iter){
            if(nums[iter]!=nums[index] || nums[iter]!=nums[
                index-1]){
                nums[++index]=nums[iter];
            }
        }
        return ++index;
    }
};
```

**Code 2\*: another include implementation**

This is more concise code for include implementation. Time complexity is  $O(n)$ , and space complexity is  $O(1)$ . Also, the following code is easy to generate to allow at most  $k$  duplicates.

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int length = 0;
        for (int n : nums) {
            if (length < 2 || n > nums[length-2])
                nums[length++] = n;
        }
        return length;
    }
};
```

**Code 3: preclude implementation**

Time complexity is  $O(n)$ , and space complexity is  $O(1)$ .

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int index=0;
        const unsigned long n = nums.size();
        for(unsigned iter=0; iter<n; ++iter){
            if(iter>0 && iter<n-1 && nums[iter]==nums[iter-1] && nums[iter]
                ==nums[iter+1])
                continue;
            nums[index++]=nums[iter];
        }
    }
};
```

```

    }
    return index;
}
};

```

## Related Problems

- Remove Duplicates from Sorted Array. See section 1.1.6.

### 1.1.8 Search in Rotated Sorted Array

#### Description

(33 medium) Suppose an array sorted in ascending order 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.

#### Analysis

The key of solving these problems is to find the unique feature of the array. Using the idea of binary search, define *first*, *last*, *mid*, and we should notice that if `nums[first] < nums[last]`, it is a sorted subarray, otherwise, it is a rotated subarray. We also need to be very careful on the boundary conditions of this problem. We need to be sure that the length of the array is contracting after each step even there are 2 elements left in an array.

#### Code

```

class Solution {
public:
    int search(vector<int>& nums, int target) {
        unsigned long first=0;
        unsigned long last=nums.size();
        unsigned long mid;
        while(first < last){
            mid=first + (last-first)/2;
            if(nums[mid]==target) return (int)mid;
            if(nums[first] < nums[mid]){
                if( nums[first] <= target && target <
                    nums[mid]){
                    last = mid;
                } else {
                    first =mid+1;
                }
            }
        }
    }
}

```

```

        } else {
            if(nums[mid]<target && target <= nums[
                last-1]){
                first = mid+1;
            } else {
                last = mid;
            }
        }
    }
    return -1;
}
};

```

Because the way that we write the `while (last > first)` loop, we can save one line code at beginning to check if the vector is empty, i.e. `if (last == 0) return -1;`

### 1.1.9 Search in Rotated Sorted Array II

#### Description

(81 medium) Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed? Would this affect the run-time complexity? How and why? Suppose an array sorted in ascending order 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`). Write a function to determine if a given target is in the array. The array may contain duplicates.

#### Analysis

The key to success in solving searching in rotated sorted array without duplicate example (1.1.6) is the recognition of sorted subarray vs rotated subarray by testing the condition of `nums[first] < nums[last]`. When duplicates exist, cases as `{1,1,1,3,1}` has to be taken care of. A simple solution to build on previous result is to skip the duplicate once we find that `nums[first] == nums[mid]`.

#### Code

```

class Solution {
public:
    bool search(vector<int>& nums, int target) {
        unsigned long first=0;
        unsigned long last=nums.size();
        unsigned long mid;
        while(last > first){
            mid=first + (last-first)/2;
            if(nums[mid]==target) return true;
            if(nums[first] < nums[mid]){

```

```

        if( nums[first] <= target && target <
nums[mid]){
            last = mid;
        } else {
            first =mid+1;
        }
    } else if (nums[first] > nums[mid]){
        if(target > nums[mid] && target <= nums[
last-1]){
            first = mid+1;
        } else {
            last = mid;
        }
    } else {
        first ++;
    }
}
return false;
}
};

```

### 1.1.10 Median of Two Sorted Arrays

#### Description

(4 hard) There are two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively. Find the median of the two sorted arrays. The overall run time complexity should be  $O(\log(m+n))$ .

Example 1: `nums1 = [1, 3]; nums2 = [2]`. The median is 2.0.

Example 2: `nums1 = [1, 2]; nums2 = [3, 4]`. The median is  $(2+3)/2 = 2.5$ .

#### Analysis

A general problem is to find the  $k^{th}$  item in two sorted array. We can solve this problem recursively. The simple idea starts from remove one item from one of two arrays at each time but the time complexity will be  $O(m+n)$ . Since the two arrays are sorted, we can remove much more than one each time. Actually, we can remove  $k/2$  items. Then we pay special attention on stopping conditions for the recursive calls.

#### Code

```

class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<
int>& nums2) {
        unsigned long totalN = nums1.size()+nums2.size();
    }
};

```

```

        if(totalN%2==0){
            return (findKth(nums1.begin(),nums1.size(), nums2.
                begin(),nums2.size(),totalN/2)+findKth(nums1.begin(),
                nums1.size(), nums2.begin(),nums2.size(),totalN/2+1))
                /2.0;
        } else {
            return (findKth(nums1.begin(),nums1.size(), nums2.
                begin(),nums2.size(),(totalN+1)/2) );
        }
    }
private:
int findKth(std::vector<int>::iterator iterA, unsigned long m,
std::vector<int>::iterator iterB, unsigned long n, unsigned long
k){
    if(m==0) return *(iterB+k-1);
    if(m > n) return findKth(iterB,n, iterA, m, k);
    if(k==1) return min(*iterA,*iterB);
    unsigned long reduceA=min(k/2,m);
    unsigned long reduceB=k-reduceA;
    if(*(iterA+reduceA-1) < *(iterB+reduceB-1)){
        return findKth(iterA+reduceA, m-reduceA, iterB, n, k-
            reduceA);
    } else if(*(iterA+reduceA-1) > *(iterB+reduceB-1)) {
        return findKth(iterA,m, iterB+reduceB,n-reduceB, k-
            reduceB);
    } else {
        return *(iterA+reduceA-1);
    }
}
};

```

### 1.1.11 Longest Consecutive Sequence

#### Description

(128 hard) Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

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.

#### Analysis

The simplest solution is to sort the array first, but time complexity is at least  $n \log(n)$ . One approach is to use an `unordered_set` to store the values and search in this hash table. Once a number is visited as in the consecutive sequence, we should remove it from the set. Otherwise, the code is inefficient due to duplicated visits. Another version is to use `unordered_map<int, bool>`

to store value and whether this value has been evaluated as consecutive sequence before. To avoid duplicated visit, another idea is to start with the minimal number in a sequence. We implement these three ideas as below.

### Code: using unordered\_set

The following code is  $O(n)$

```
class Solution {
public:
    int longestConsecutive(vector<int> &num) {
        unordered_set<int> record(num.begin(), num.end());
        int res = 1;
        for(int n : num){
            if(record.find(n)==record.end()) continue;
            record.erase(n);
            int prev = n-1, next = n+1;
            while(record.find(prev)!=record.end()) record.erase(prev--);
            while(record.find(next)!=record.end()) record.erase(next++);
            res = max(res, next-prev-1);
        }
        return res;
    }
};
```

### Code: using unordered\_map

This code has similar idea of using unordered\_map<int, bool> without erasing elements in the set.

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        int result=0;
        unordered_map<int, bool> map;
        for(auto i:nums){
            map.insert(make_pair(i, false));
        }
        for(auto iter=map.begin(); iter!=map.end(); ++iter){
            if(iter->second) continue;
            int center_up = iter->first;
            int center_down = iter->first;
            int length = 1;
            while(map.find(++center_up)!=map.end()){
                map[center_up]=true;
                length++;
            }
            while(map.find(--center_down)!=map.end()){
                map[center_down]=true;
            }
        }
        return result;
    }
};
```

```

        length++;
    }
    if (length > result) result = length;
}
return result;
}
};

```

### Code\*: starting from minimal number

```

class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> s(nums.begin(), nums.end());
        int res = 0;
        for (int n : s) {
            if (s.find(n - 1) == s.end()) {
                int m = n + 1;
                while (s.find(m) != s.end())
                    m++;
                res = max(res, m - n);
            }
        }
        return res;
    }
};

```

## 1.1.12 Container With Most Water

### Description

(11 medium) Given  $n$  non-negative integers  $a_1, a_2, \dots, a_n$ , where each represents a point at coordinate  $(i, a_i)$ .  $n$  vertical lines are drawn such that the two endpoints of line  $i$  is at  $(i, a_i)$  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 and  $n$  is at least 2.

### Analysis

This problem has some ambiguity. Based on the description, we only need two vertical lines to form a container. It ignores any other lines in between. A key feature of the solution is that if  $a_i$  and  $a_j$  are the two lines ( $i < j$ ) to form the largest container, there will be no taller bars outside  $a_i$  and  $a_j$ . Otherwise, it contradicts the fact that  $a_i$  and  $a_j$  form the largest container. Start by evaluating the widest container, using the first and the last line. All other possible containers are less wide, so to hold more water, they need to be higher. Thus, after evaluating that widest container, skip lines at both ends that don't support a higher height. Then evaluate that new container we arrived at. Repeat until there are no more possible containers left.



**Code**

```

class Solution {
public:
    int maxArea(vector<int>& height) {
        int area=0;
        unsigned long i=0, j=height.size()-1;
        int h=0;
        while(i<j){
            h = min(height.at(j),height.at(i));
            area=max(area,int(j-i)*h);
            while(height.at(i) <= h && i<j) ++i;
            while(height.at(j) <= h && i<j) --j;
        }
        return area;
    }
};

```

**1.1.13 Search Insert Position****Description**

(35 easy) 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.

Example 1: Input: [1, 3, 5, 6], 5. Output: 2.

Example 2: Input: [1, 3, 5, 6], 2. Output: 1.

Example 3: Input: [1, 3, 5, 6], 7. Output: 4.

Example 4: Input: [1, 3, 5, 6], 0. Output: 0.

**Analysis**

We use binary search with time complexity as  $O(\log n)$ .

**Code**

```

class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int low = 0, high = nums.size()-1;
        while (low <= high) {
            int mid = low + (high-low)/2;
            if (nums[mid] < target)
                low = mid+1;
            else

```

```

        high = mid-1;
    }
    return low;
}
};

```

These codes are very concise. Please pay attention to how it handle various cases in a unified framework. We should also notice that we use `mid = low + (high-low)/2;` instead of `mid = (low+high)/2`, because the later could overflow when `low` and `high` are large numbers.

### 1.1.14 Plus One

#### Description

(66 easy) Given a non-negative integer represented as a non-empty array of digits, plus one to the integer. You may assume the integer do not contain any leading zero, except the number 0 itself. The digits are stored such that the most significant digit is at the head of the list.

#### Analysis

We provided three implementation below. The first approach uses a reverse iterator. It add one first, then correct the carries in each position. The first implementation uses an insert method to add 1 at beginning which may not be efficient. The second algorithm improve its efficiency by pushing back a 0 into the vector. The last algorithm make the loop more concise. The signature of this problem is not well defined. It should either pass into a constant reference, or have a void return, not both.

#### Code 1:

```

class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        ++digits.back();
        for(auto i=digits.rbegin(); i!=digits.rend()-1; ++i) {
            if(*i<=9) break;
            else
            { *i=0;
              ++*(i+1);
            }
        }
        if(*digits.begin()==10) {
            *digits.begin()=0;
            digits.insert(digits.begin(), 1);
        }
        return digits;
    }
};

```

**Code\* 2:**

```

class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        for(auto i=digits.rbegin();i!=digits.rend();++i){
            if(*i==9){
                *i=0;
            } else {
                ++*i;
                return digits;
            }
        }
        *digits.begin()=1;
        digits.push_back(0);
        return digits;
    }
};

```

**Code 3:**

```

class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        for (size_t i=digits.size(); i-->0; digits[i] = 0)
            if (digits[i]++ < 9)
                return digits;
        digits[0]++;
        digits.push_back(0);
        return digits;
    }
};

```

**1.1.15 Climbing Stairs****Description**

(70 easy) You are climbing a stair case. It takes  $n$  steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? Note: Given  $n$  will be a positive integer.

Example 1: Input: 2 Output: 2 Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

Example 2: Input: 3 Output: 3 Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

### Analysis

We can solve this problem recursively, but it exceeds time limit. We notice that if we know solution for  $n-1$  and  $n-2$  step, the choices of moving to  $n$  steps is the summation of them. Therefore, we can use previous results. This problem illustrates a way to make recursive calls more efficient, i.e. memorization. We further observed that, we only need to use the results from the previous two steps. Therefore, we provided a more concise solution in Code 3. Variable  $a$  tells you the number of ways to reach the current step, and  $b$  tells you the number of ways to reach the next step. So for the situation one step further up, the old  $b$  becomes the new  $a$ , and the new  $b$  is the old  $a$ + old  $b$ , since that new step can be reached by climbing 1 step from what  $b$  represented or 2 steps from what  $a$  represented.

### Code 1: recursive calls

```
class Solution {
    int c{0};
    public:
    int climbStairs(int n) {
        countChoices(n);
        return c;
    }
    void countChoices(int n){
        if(n<2){
            ++c;
            return;
        }
        else{
            countChoices(n-1);
            countChoices(n-2);
        }
    }
};
```

### Code 2\*: dynamic programming

```
class Solution {
    public:
    int climbStairs(int n) {
        if(n==1) return 1;
        if(n==2) return 2;
        vector<int> choices(n, 0);
        choices.at(0)=1;
        choices.at(1)=2;
```

```

        for(size_t i=2;i!=n;++i){
            choices.at(i)=choices.at(i-1)+choices.at(i-2);
        }
        return choices.back();
    }
};

```

**Code 3**

```

class Solution {
public:
    int climbStairs(int n) {
        int a = 1, b = 1;
        while (n--)
            a = (b += a) - a;
        return a;
    }
};

```

***1.1.16 Merge Two Sorted Arrays*****Description**

(88 easy) Given two sorted integer arrays `nums1` and `nums2`, merge `nums2` into `nums1` as one sorted array.

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

**Analysis**

The key to solve this problem is to sort from the largest value first so that we do not need to move the objects around. We should also pay attention to how to handle the cases where there are some `nums2` or `nums1` left after sorting from the largest.

**Code**

```

class Solution {
public:
    void merge(vector<int> &nums1, int m, vector<int> &nums2,
        int n) {
        for (int i = m - 1, j = n - 1, k = m + n - 1; j >=
            0;)

```

```

        nums1[k--] = i >= 0 && nums1[i] > nums2[j] ? nums1[i]
        : nums2[j--];
    }
};

```

## 1.2 Singly Linked List

The definition of single linked list is as below,

```

struct ListNode {
    int val;
    ListNode *next;
    explicit ListNode(int x) : val(x), next(nullptr) {}
};

```

### 1.2.1 Add Two Numbers

#### Description

(2 medium) You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list. You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example: Input: (2->4->3) + (5->6->4), Output: 7->0->8.

#### Analysis

We need to pay special attention for the cases where there are different lengths of the lists, and cases where the result has more digits than each of the array, such as  $10=7+3$ .

#### Code

```

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode result(-1);
        ListNode *current= &result;
        int carry = 0;
        while(l1 || l2 || carry){
            int value = (l1?l1->val:0)+(l2?l2->val:0)+carry;
            ;
            current->next = new ListNode(value%10);

```

```

        carry = value/10;
        l1=l1?l1->next: nullptr;
        l2=l2?l2->next: nullptr;
        current=current->next;
    }
    return result.next;
};

```

We should notice that `nullptr` can be casted into `bool` variable for conditional statement.

### 1.2.2 Merge Two Sorted Arrays

#### Description

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

#### Analysis

We are not necessary to create a new singly linked list. We can just use the existing linked list.

#### Code

```

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode result(INT_MIN);
        ListNode * current=&result;
        for(;l1!= nullptr && l2!= nullptr;current=current->
next){
            if(l1->val<l2->val){
                current->next=l1;
                l1=l1->next;
            } else{
                current->next=l2;
                l2=l2->next;
            }
        }
        current->next = l1==nullptr?l2:l1;
        return result.next;
    }
};

```

We first deal with the cases that `l1!= nullptr && l2!= nullptr`. The cases where either or both are empty can be combined. Also, we can use the `for`-loop to make the code more concise.

### 1.2.3 Remove Duplicates from Sorted List

#### Description

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

Example 1: Given  $1 \rightarrow 1 \rightarrow 2$ , return  $1 \rightarrow 2$ .

Example 2: Given  $1 \rightarrow 2 \rightarrow 3 \rightarrow 3$ , return  $1 \rightarrow 2 \rightarrow 3$ .

#### Analysis

This problem can be solved by iterations or by recursion.

#### Code\*

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        auto current = head;
        while(current) {
            while(current->next && current->val==current->
                next->val) {
                delete current->next;
                current->next=current->next->next;
            }
            current=current->next;
        }
        return head;
    }
};
```

The above codes can handle `head==nullptr` case naturally.

#### Code: recursive solution

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head && (head->next = deleteDuplicates(head->next
        )) && head->next->val == head->val) {
            auto oldhead=head;
            head = head->next;
            delete oldhead;
        }
        return head;
    }
};
```



The key to solve this problem recursively is to reduce the original problem through `head->next = deleteDuplicates(head->next)`.



## Chapter 2

### String

#### 2.1 Longest Substring Without Repeating Characters

##### Description

(3 medium) Given a string, find the length of the longest substring without repeating characters.

Example 1: Given "abcabcbb", the answer is "abc", which the length is 3.

Example 2: Given "bbbb", the answer is "b", with the length of 1.

Example 3: Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

##### Analysis

We can have a moving substring with a start position, and a head. The head moves forward. When it reads a new character, we check when this new character shows up last time. If it is after the start position, it means that the current substring contains this character, then we change the start position. The following solution has time complexity  $O(n)$  and space complexity  $O(1)$ .

##### Code

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> lastShow(256, -1);
        int start=0;
        int max_len=0;
        for(int i=0; i!=s.size(); ++i) {
            if(lastShow[s[i]]>=start) {
                max_len=max(max_len, i-start);
                start=lastShow[s[i]]+1;
            }
            lastShow[s[i]]=i;
        }
    }
};
```

```

    }
    return max(max_len, (int)s.size()-start);
}
};

```

## 2.2 Palindrome Number

### Description

(9 easy) Determine whether an integer is a palindrome. Do this without extra space.

Some hints: Could negative integers be palindromes? (ie, -1) If you are thinking of converting the integer to string, note the restriction of using extra space. You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case? There is a more generic way of solving this problem.

### Analysis

Although this is not a string question, the solutions can be used to solve string related problems. We should not reverse the integer since it might overflow. One approach is to compare the first and last digits recursively. Another approach is reverse to the digits half way then compare it. We should be careful on the later solution to handle some integer like 10.

### Code\*

```

class Solution {
public:
    bool isPalindrome(int x) {
        if(x<0) return false;
        int d=1;
        while(x/d>=10) d*=10;
        while(x>0){
            if(x/d!=(x%10)) return false;
            x=x%10;
            d/=100;
        }
        return true;
    }
};

```

### Code: reversing till half

```

class Solution {

```

```

public:
bool isPalindrome(int x) {
    if (x<0 || (x!=0 &&x%10==0)) return false;
    int sum=0;
    while (x>sum)
    {
        sum = sum*10+x%10;
        x = x/10;
    }
    return (x==sum) || (x==sum/10);
}
};

```

## 2.3 Longest Palindromic Substring

### Description

(4 medium) Given a string  $s$ , find the longest palindromic substring in  $s$ . You may assume that the maximum length of  $s$  is 1000.

Example 1: Input: "babad". Output: "bab". Note: "aba" is also a valid answer.

Example 2: Input: "cbabd". Output: "bb"

### Analysis

This is a classical problem and we have the following solutions.

**Brutal force:** If a string length is  $n$ , there are  $C(n, 2)$  substrings. We can check each of these substrings and report the longest palindromic substring. This solution has  $O(n^3)$  time complexity.

**Center-expansion I:** Another approach is to consider each of the character as a center of a palindromic substring and search all of the characters in a string. Because the center can have repeat characters such as *baaab*. We need to identify the center first, then expand to search boundaries. This solution has  $O(n^2)$  time complexity.

**Center-expansion II:** We observe that a palindrome mirrors around its center. Therefore, a palindrome can be expanded from its center, and there are only  $2n - 1$  such centers. The reason is the center of a palindrome can be in between two letters. Such palindromes have even number of letters (such as "abba") and its center are between the two bs. We can add some dummy character in the string to make the code easy as changing "abba" into "a#b#b#a". Adding '#' is a concept, and we do not really need to add it in the program. Time complexity of this solution is  $O(n^2)$ .

**Dynamic programming:** Dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization". It following the 3 steps: define success, build a recurrence formula, and define base cases:

1. define a look up table for previous solution, `table[i][j]=true` iff substring `s[i]` to `s[j]` is a palindromic substring.
2. `table[i-1][j+1]=true` iff `table[i][j]=true`.
3. build up the base case as `table[i][i]=true` and `table[i][i+1]=true` if `s[i] = s[i+1]`.

This solution has  $O(n^2)$  time complexity and  $O(n^2)$  space complexity.

Manachers Algorithm: Here, we discuss an algorithm that runs in  $O(N)$  time and  $O(N)$  space, also known as Manachers algorithm. Further explanations can be found from <https://articles.leetcode.com/longest-palindromic-substring-part-ii/>

### Code\*: center-expansion algorithm I

Time complexity is  $O(n^2)$ .

```
class Solution {
public:
    string longestPalindrome(string s) {
        if(s.size()<2) return s;
        string result;
        for(unsigned j=0; j<s.size(); ++j) {
            unsigned i=j, k=j;
            if((s.size()-1-j)< result.size()/2) break;
            while(k<s.size()-1 && s[k+1]==s[k]) ++k;
            while(i>0 && k<s.size()-1 && s[i-1]==s[k+1]) --i, ++k;
            result= result.size()<(k-i+1)?s.substr(i, (k-i+1)):result;
        }
        return result;
    }
};
```

### Code: center-expansion algorithm II

Time complexity is  $O(n^2)$ .

```
class Solution {
public:
    string longestPalindrome(string s) {
        if(s.size()<2) return s;
        unsigned long n=2*s.size()-1;
        string result;
        for(unsigned j=0; j<n; ++j) {
            if(j%2==0) {
                unsigned i=j/2, k=j/2;
                while(i>0 && k<s.size()-1 && s[i-1]==s[k+1]) --i, ++k;
                result= result.size()<(k-i+1)? s.substr(i, k-i+1):result;
            } else {
```

```

        unsigned i=j/2, k=(j+1)/2;
        if(s[i]!=s[k]) continue;
        while(i>0 && k<s.size()-1 && s[i-1]==s[k+1]) --
            i,++k;
        result= result.size()<(k-i+1)? s.substr(i,k-i
            +1):result;
    }
}
return result;
}
};

```

### Code: dynamic programming

```

class Solution {
public:
    string longestPalindrome(string s) {
        if(s.size()<2) return s;
        unsigned maxLen = 1, begin=0;
        bool table[1000][1000]{false};
        for(unsigned i=0;i<s.size();++i){
            table[i][i]=true;
        }
        for(unsigned i=0;i<s.size()-1;++i){
            if(s[i]==s[i+1]){
                table[i][i+1]=true;
                maxLen=2;
                begin=i;
            }
        }
        for(unsigned len=3; len < s.size()+1;++len){
            for(unsigned i=0;i<s.size()-len+1;++i){
                if(s[i]==s[i+len-1] && table[i+1][i+len-2]){
                    table[i][i+len-1]=true;
                    maxLen=len;
                    begin=i;
                }
            }
        }
        return s.substr(begin,maxLen);
    }
};

```

### Code: Manachers Algorithm

```

class Solution {
public:

```

```

string longestPalindrome(string s)
{
    string T;// Transform S to T
    for(int i=0;i<s.size();i++)
        T+=" "+s.substr(i,1);
    T.push_back('#');
    vector<int> P(T.size(),0); // Array to record longest
    palindrome
    int center=0,boundary=0,maxLen=0,resCenter=0;
    for(int i=1;i<T.size()-1;i++) {
        int iMirror=2*center-i; // calc mirror i =
        center-(i-center)
        P[i]=(boundary>i)?min(boundary-i,P[iMirror]):0;
        // shortcut
        while(i-1-P[i]>=0&& i+1+P[i]<=T.size()-1&&T[i+1+
        P[i]] == T[i-1-P[i]]) // Attempt to expand
        palindrome centered at i
        P[i]++;
        if(i+P[i]>boundary) { // update center and
        boundary
            center = i;
            boundary = i+P[i];
        }
        if(P[i]>maxLen) { // update result
            maxLen = P[i];
            resCenter = i;
        }
    }
    return s.substr((resCenter - maxLen)/2, maxLen);
}
};

```

## 2.4 ZigZag Conversion

### Description

(6 medium) The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility),

P		A		H		N
A	P	L	S	I	I	G
Y		I		R		

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



## Analysis

The following is another example for zigzag pattern when  $n = 4$ .

P			I			N
A		L	S		I	G
Y	A		H	R		
P			I			

The key of this algorithm is to find the corresponding relationship between the original array with the transformed matrix containing the zigzag pattern. Let  $i$  be the row index,  $j$  be the column index of the matrix, and  $n$  be the input of number of rows. It is easy to see that if there is a character in position  $(i, j)$  and it also belongs to a vertical line, its position in the string is  $(2n-2) \cdot j / (n-1) + i$ , and if it an element belongs to diagonal line, its position in the string is  $(2n-2) \cdot \{j / (n-1) + 1\} - i$ . Therefore, a direct implementation is to translate the matrix into a new string. After further observation, we can do better. We have two types of rows, when  $i == 0$  or  $i == n-1$ , i.e. the first and last row, we only have vertical elements. For other rows, there have both diagonal and vertical elements. Furthermore, we can consider every  $n-1$  columns as a unit. Therefore, for each row in each unit, there are one vertical element and at most one diagonal element. Based on this idea, we have our code below which has  $O(n)$  time complexity and  $O(1)$  space complexity.

## Code

```
class Solution {
public:
    string convert(string s, int numRows) {
        if (numRows < 2 || s.size() < 3 || numRows >= s.size()) return s;
        string result{};
        unsigned long index;
        for (unsigned i = 0; i < numRows; ++i) {
            for (unsigned j = 0;; ++j) {
                index = (2 * numRows - 2) * j + i;
                if (index < s.size()) result += s[index];
                else break;
                if (i == 0 || i == numRows - 1) continue;
                index = (2 * numRows - 2) * (j+1) - i;
                if (index < s.size()) result += s[index];
                else break;
            }
        }
        return result;
    }
};
```

## 2.5 String to Integer (atoi)

### Description

(8 medium) Implement atoi to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

Requirements for atoi: The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value. The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function. If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed. If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, `INT_MAX` (2147483647) or `INT_MIN` (-2147483648) “is returned.

### Analysis

One challenge of this problem is to handle overflow. A simple solution is to define a `long` type integer, then compare it with `INT_MAX` or `INT_MIN`. To make our code more generally applicable, we avoid that implementation.

### Code

```
class Solution {
public:
    int myAtoi(string str) {
        int sign = 1, base = 0, i = 0;
        while (str[i] == ' ') ++i;
        if (str[i] == '-' || str[i] == '+') {
            sign = (str[i++] == '-') ? -1 : 1;
        }
        while (str[i] >= '0' && str[i] <= '9') {
            if (base > INT_MAX / 10 || (base == INT_MAX /
                10 && str[i] - '0' > 7)) {
                if (sign == 1) return INT_MAX;
                else return INT_MIN;
            }
            base = 10 * base + (str[i++] - '0');
        }
        return base * sign;
    }
};
```

A special detail of this implementation is how we handle the case where the first character is non-meaningful character, such as "c12".

## 2.6 Regular Expression Matching

### Description

'.' Matches any single character. '\*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be: `bool isMatch(const char *s, const char *p)`

A few examples,

- `isMatch("aa","a") → false`
- `isMatch("aa","aa") → true`
- `isMatch("aaa","aa") → false`
- `isMatch("aa","a*") → true`
- `isMatch("aa",".*") → true`
- `isMatch("ab",".*") → true`
- `isMatch("aab","c*a*b") → true`

### Analysis

The last case is true because: '\*' Matches **zero** or more of the preceding element. This problem has a typical solution using dynamic programming. We define the state  $P[i][j]$  to be true if  $s[0..i)$  matches  $p[0..j)$  and false otherwise. Then the state equations are:

1.  $P[i][j] = P[i-1][j-1]$ , if  $p[j-1] \neq '*'$  &&  $(s[i-1] == p[j-1] \mid p[j-1] == '.')$ ;
2.  $P[i][j] = P[i][j-2]$ , if  $p[j-1] == '*'$  and the pattern repeats for 0 times;
3.  $P[i][j] = P[i-1][j] \&\& (s[i-1] == p[j-2] \mid p[j-2] == '.')$ , if  $p[j-1] == '*'$  and the pattern repeats for at least 1 times.

### Code: dynamic programming

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.length(), n = p.length();
        vector<vector<bool>> > dp(m + 1, vector<bool> (n + 1, false));
        dp[0][0] = true;
        for (int i = 0; i <= m; ++i)
            for (int j = 1; j <= n; ++j)
                if (p[j-1] == '*')
                    dp[i][j] = dp[i][j-2] || (i > 0 && (s[i-1] == p[j-2] || p[j-2] == '.')) && dp[i-1][j]);
    }
};
```

```

        else dp[i][j] = i > 0 && dp[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '.');
        return dp[m][n];
    }
};

```

## 2.7 Integer to Roman

### Description

(12 medium) Given an integer, convert it to a roman numeral. Input is guaranteed to be within the range from 1 to 3999

### Analysis

We can consider to use pair and vector to make the code easy to debug.

### Code

```

class Solution {
public:
    string intToRoman(int num) {
        if(num<1 || num > 3999) return {};
        string result;
        vector<pair<int,string>> iToR{make_pair(1000,"M"),
make_pair(900,"CM"),make_pair(500,"D"),make_pair
(400,"CD"),make_pair(100,"C"),make_pair(90,"XC"),
make_pair(50,"L"),make_pair(40,"XL"), make_pair(10,"X"),
make_pair(9,"IX"),make_pair(5,"V"),make_pair(4,"
IV"),make_pair(1,"I")};
        for(auto x:iToR){
            for(int i=0; i<num/x.first;++i){
                result+=x.second;
            }
            num%=x.first;
        }
        return result;
    }
};

```

We use += instead of insert. Also, we use num%=x.first as a simple way to get the reminders.

## 2.8 Roman to Integer

### Description

(13 easy) Given a roman numeral, convert it to an integer. Input is guaranteed to be within the range from 1 to 3999.

### Analysis

### Code

```
class Solution {
public:
    int romanToInt(string s) {
        map<char,int> rToI{ { 'I' , 1 },
                           { 'V' , 5 },
                           { 'X' , 10 },
                           { 'L' , 50 },
                           { 'C' , 100 },
                           { 'D' , 500 },
                           { 'M' , 1000 } };
        int result=0;
        for(auto i=s.begin();i!=s.end();++i){
            if(i+1!=s.end()){
                result= rToI[*i+1]<=rToI[*i]? result+
                    rToI[*i]:result-rToI[*i];
            } else {
                result += rToI[*i];
            }
        }
        return result;
    }
};
```

To initiate a map, we can either use `make_pair` or use `{}`.

## 2.9 Longest Common Prefix

### Description

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

### Analysis

We can scan the string of string horizontally or vertically.

**Code\*: vertical scan**

```

class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if(strs.empty()) return{};
        char x;
        for(unsigned index=0; index<strs.at(0).size(); ++index)
        {
            x=strs.at(0).at(index);
            for(auto str=strs.begin()+1; str!=strs.end(); ++str)
            {
                if(index>str->size()-1 || (*str)[index]
                    != x){
                    return strs.at(0).substr(0, index);
                }
            }
        }
        return strs.at(0).substr(0, strs.at(0).size());
    }
};

```

**Code: horizontal scan**

```

class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if(strs.empty()) return {};
        unsigned long rBound = strs.at(0).size();
        for(auto x=strs.begin(); x!=strs.end(); ++x)
        {
            rBound=min(rBound, x->size());
            for(unsigned i=0; i<rBound; ++i)
            {
                if(strs.at(0).at(i) != (*x).at(i))
                {
                    rBound=i;
                    break;
                }
            }
        }
        return strs.at(0).substr(0, rBound);
    }
};

```

## 2.10 Implement strStr()

### Description

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

Example 1: Input: haystack = "hello", needle = "ll". Output: 2.

Example 2: Input: haystack = "aaaaa", needle = "bba". Output: -1

### Analysis

The simplest solution is to use STL. We can also use a brutal force to search which has a time complexity  $O(mn)$ . More efficient algorithms include KMP algorithm, Boyer-Mooer algorithm, and Rabin-Karp algorithm. We provided the KMP algorithms code below, and the explanation can be found online <http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/>.

### Code\*: using STL

```
class Solution {
public:
    int strStr(string haystack, string needle) {
        return (int)haystack.find(needle);
    }
};
```

### Code: brutal force

```
class Solution{
public:
    int strStr(string haystack, string needle) {
        if(needle.empty()) return 0;
        const int diff = int(haystack.size())-int(needle.size())+1;
        for(int i=0; i<diff;++i){
            for(int j=0; j<needle.size();++j){
                if(haystack[i+j]!=needle[j]) break;
                if(j==needle.size()-1) return i;
            }
        }
        return -1;
    }
};
```

We should be careful here to use `haystack.size()-needle.size()` because both of them are unsigned type. So when `haystack.size()<needle.size()`, the difference will be overflowed.

**Code: KMP algorithm**

```

class Solution {
public:
    int strStr(string haystack, string needle) {
        int m = haystack.length(), n = needle.length();
        if (!n) return 0;
        vector<int> lps = kmpProcess(needle);
        for (int i = 0, j = 0; i < m; ) {
            if (haystack[i] == needle[j]) {
                i++;
                j++;
            }
            if (j == n) return i - j;
            if (i < m && haystack[i] != needle[j]) {
                if (j) j = lps[j - 1];
                else i++;
            }
        }
        return -1;
    }
private:
    vector<int> kmpProcess(string& needle) {
        int n = needle.length();
        vector<int> lps(n, 0);
        for (int i = 1, len = 0; i < n; ) {
            if (needle[i] == needle[len])
                lps[i++] = ++len;
            else if (len) len = lps[len - 1];
            else lps[i++] = 0;
        }
        return lps;
    }
};

```

**2.11 Count and Say**

(38 easy) The count-and-say sequence is the sequence of integers with the first five terms as following:

1. 1
2. 11
3. 21
4. 1211
5. 111221

- 1 is read off as “one 1” or 11.
- 11 is read off as “two 1s” or 21.
- 21 is read off as “one 2, then one 1” or 1211.

Given an integer  $n$ , generate the  $n$ th term of the count-and-say sequence.

Note: Each term of the sequence of integers will be represented as a string.



Example 1: Input: 1. Output: "1".

Example 2: Input: 4. Output: "1211"

### Analysis

(27 easy) To say the  $n^{\text{th}}$  sequence, we need to know the  $(n-1)^{\text{th}}$  sequence. We can solve the problem recursively.

### Code

```
class Solution {
public:
    string countAndSay(int n) {
        if(n<1) return{};
        if(n==1) return "1";
        return sayString(countAndSay(n-1));
    }
    string sayString(const string &s) {
        string ss;
        for (auto i = s.begin(); i != s.end(); ) {
            auto j = find_if(i, s.end(), bind1st(
                not_equal_to<char>(), *i));
            ss+= to_string(distance(i, j)) + *i;
            i = j;
        }
        return ss;
    }
};
```

These codes also show how to use STL to handle string.

## 2.12 Maximum Subarray

### Description

(53 easy) Find the contiguous subarray within an array (containing at least one number) which has the largest sum. For example, given the array `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the contiguous subarray `[4, -1, 2, 1]` has the largest sum=6. More practice: If you have figured out the  $O(n)$  solution, try coding another solution using the divide and conquer approach, which is more subtle.

### Analysis

We observe that the sum of the subarray is increasing if and only if the previous sum is greater than zero. Therefore, we can keep adding each integer to the sequence until the sum drops below 0. If

sum is negative, then should reset the sequence. One approach is to have two iterator pointing the begin and end of a subarray. After we further observe the pattern, we can simplify it as below.

### Code: original solution

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if (nums.empty()) return 0;
        int mv=INT_MIN, t=INT_MIN;
        for(auto i=nums.begin(); i!=nums.end(); ){
            auto j=i+1;
            t=*i;
            mv=t>mv?t:mv;
            while(j!=nums.end() && t>0){
                t+=*j;
                mv=t>mv?t:mv;
                ++j;
            }
            i=j;
        }
        return mv;
    }
};
```

### Code: simplified

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if (nums.empty()) return 0;
        int sum=0, maxV=nums[0];
        for(auto x:nums){
            sum+=x;
            maxV=max(maxV, sum);
            sum=max(0, sum);
        }
        return maxV;
    }
};
```

### Code\*: further simplified

Previously solution can be further simplified as below.

```
class Solution {
public:
```

```

int maxSubArray(vector<int>& nums) {
    if (nums.empty()) return 0;
    int sum=0,maxV=nums[0];
    for(auto x:nums){
        sum=max(sum+x,x);
        maxV=max(maxV,sum);
    }
    return maxV;
};

```

## 2.13 Length of Last Word

### Description

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

Example 1: Input: "Hello World". Output: 5.

### Analysis

We need to pay attention on strings ending with spaces, such as "Hello ".

### Code

```

class Solution {
public:
    int lengthOfLastWord(string s) {
        auto x=s.rbegin();
        while(*x==' ') ++x;
        auto y=find(x,s.rend()),' ');
        return distance(x,y);
    }
};

```

## 2.14 Add Binary

### Description

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

Example 1: Input: a = "11", b = "1". Return: "100".

## Analysis

The string a and b can be in different sizes, and pay attention to how we can treat them uniformly as below.

## Code

```
class Solution
{
public:
    string addBinary(string a, string b)
    {
        string s = "";
        int c = 0, i = a.size() - 1, j = b.size() - 1;
        while(i >= 0 || j >= 0 || c == 1)
        {
            c += i >= 0 ? a[i --] - '0' : 0;
            c += j >= 0 ? b[j --] - '0' : 0;
            s = char(c % 2 + '0') + s;
            c /= 2;
        }
        return s;
    }
};
```

## Chapter 3

# Tree

The definition of binary tree is as below.

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

### 3.1 Binary Tree Traversals

#### 3.1.1 Same Tree

##### Description

(100 easy) Given two binary trees, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

Example 1:    Input:     $\begin{array}{c} 1 \\ / \quad \backslash \\ 2 \quad 3 \end{array}$     and     $\begin{array}{c} 1 \\ / \quad \backslash \\ 2 \quad 3 \end{array}$     Output: true.

Example 2:    Input:     $\begin{array}{c} 1 \\ / \quad \backslash \\ 2 \quad \end{array}$     ,     $\begin{array}{c} 1 \\ / \quad \backslash \\ \quad 2 \end{array}$     Output: false.

Example 3:    Input:     $\begin{array}{c} 1 \\ / \quad \backslash \\ 2 \quad 3 \end{array}$     ,     $\begin{array}{c} 1 \\ / \quad \backslash \\ 3 \quad 2 \end{array}$     Output: false.

## Analysis

We provided two ideas based on recursive and iterative algorithms. To access children nodes, the parent node should not be empty. We illustrate how to get a concise code 2 from 1 code. The iterative solution is based on an idea to put tree nodes into a stack.

### Code 1: recursive solution

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p== nullptr && q== nullptr) return true;
        if(p== nullptr ^ q== nullptr) return false;
        if(p->val!=q->val) return false;
        else{
            return isSameTree(p->left,q->left) &&
                isSameTree(p->right,q->right);
        }
    }
};
```

### Code 2\*: concise recursive solution

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (!p || !q) return q == p;
        return p->val == q->val && isSameTree(p->left, q->
            left) && isSameTree(p->right, q->right);
    }
};
```

### Code 3: Iterative solution

```
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        stack<TreeNode*> s;
        s.push(p);
        s.push(q);
        while(!s.empty()) {
            p = s.top(); s.pop();
            q = s.top(); s.pop();
            if (!p && !q) continue;
            if (!p || !q) return false;
            if (p->val != q->val) return false;
        }
    }
};
```

```

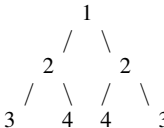
        s.push(p->left);
        s.push(q->left);
        s.push(p->right);
        s.push(q->right);
    }
    return true;
}
};

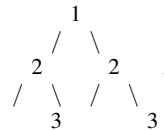
```

### 3.1.2 Symmetric Tree

#### Description

(101 easy) Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center). For example, this binary tree [1,2,2,3,4,4,3] is symmetric: Note: Bonus points if you could solve it both recursively and iteratively.

Example 1:    Input:        . Output: true

Example 2:    Input:        . Output: false

#### Analysis

For recursive solution, the key is to find a recursive pattern to make the big problem smaller and smaller. Therefore, the key is to translate the systemic information into a recursive pattern. The iterative solution is often based on a stack.

#### Code 1\*: recursive solution

```

class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return checkSymmetric(root->left, root->right);
    }
    bool checkSymmetric(TreeNode* left, TreeNode* right){
        if(!left || !right) return left==right;
        return left->val== right->val && checkSymmetric(left
->left, right->right) && checkSymmetric(left->right,
right->left);
    }
};

```

```
    }
};
```

### Code 2: iterative solution

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        stack<TreeNode*> myStack{};
        myStack.push(root->left);
        myStack.push(root->right);
        TreeNode* left{nullptr};
        TreeNode* right{nullptr};
        while(!myStack.empty()){
            left=myStack.top();
            myStack.pop();
            right=myStack.top();
            myStack.pop();
            if(!left && !right) continue;
            if(!left || !right || left->val!=right->val)
                return false;
            myStack.push(left->left);
            myStack.push(right->right);
            myStack.push(left->right);
            myStack.push(right->left);
        }
        return true;
    }
};
```

## 3.2 Construction of Binary Tree

## 3.3 Binary Search Tree

## 3.4 Recursion with a Binary Tree

### 3.4.1 *Maximum Depth of Binary Tree*

#### Description

(104 easy) Given a binary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.



**Analysis**

Since a sub-tree has the same structure as its parent tree. So that this problem can be solved recursively.

**Code**

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



## Chapter 4

# Stack and Queue

### 4.1 Stack

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. push() function is used to insert new elements into the Stack and pop() function is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

#### 4.1.1 Valid Parentheses

##### Description

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. The brackets must close in the correct order, "()" and "{}" are all valid but "]" and "([)]" are not.

##### Analysis

We solve this problem with stack data structure.

##### Code

```

class Solution {
public:
    bool isValid(string s) {
        stack<char> paren;
        for (char& c : s) {
            switch (c) {
                case '(':
                case '[':
                case '{':
                    paren.push(c); break;
                case ')': if (paren.empty() ||
                    paren.top() != '(') return false;
                    else paren.pop(); break;
                case '}': if (paren.empty() ||
                    paren.top() != '{') return false;
                    else paren.pop(); break;
                case ']': if (paren.empty() ||
                    paren.top() != '[') return false;
                    else paren.pop(); break;
                default: ; // pass
            }
        }
        return paren.empty() ;
    }
};

```

## 4.2 Queue

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called REAR(also called tail), and the deletion of existing element takes place from the other end called as FRONT(also called head). This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

## Chapter 5

# Details Implementation

### 5.1 Reverse Integer

#### Description

(7 easy) Given a 32-bit signed integer, reverse digits of an integer.

Example 1: Input: 123. Output: 321.

Example 2: Input: -123. Output: -321.

Example 3: Input: 120. Output: 21.

Note: Assume we are dealing with an environment which could only hold integers within the 32-bit signed integer range. For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

#### Analysis

The key idea to solve this problem is to have the recurrence relationship, and define a stopping condition. We also need to pay attention to the overflows.

#### Code

```
class Solution {
public:
    int reverse(int x) {
        int y=0;
        const int z= INT_MAX/10;
        while(x!=0){
            if(z < abs(y)) return y=0;
            y=y*10+x%10;
            x/=10;
        }
        return y;
    }
}
```

```
};
```

## 5.2 Sqrt(x)

### Description

(69 easy) Implement `int sqrt(int x)`. Compute and return the square root of  $x$ .  $x$  is guaranteed to be a non-negative integer.

Example 1: Input 4, output 2.

Example 2: Input 8, output 2. The square root of 8 is 2.82842..., and since we want to return an integer, the decimal part will be truncated.

### Analysis

This is a simple but tricky problem. The key is to handle the stopping and boundary conditions. We need to pay special attention on integer overflow. Another way to solve this problem is to use the Newton's method. The Newton's method also known (as the NewtonRaphson method) is a root finding algorithm:

$$x : f(x) = 0.$$

The method starts with a function  $f$  defined over the real numbers  $x$ , the function's derivative  $f'$ , and an initial guess  $x_0$  for a root of the function  $f$ . If the function satisfies the assumptions made in the derivation of the formula and the initial guess is close, then a better approximation  $x$  can be updated. In general, the process is repeated as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

until a sufficiently accurate value is reached.

Let  $x$  is the input, and  $r$  as the solution for our situation. We would like to solve an equation as  $r^2 - x = 0$ . Therefore,

$$\begin{aligned} r_{n+1} &= r_n - \frac{r^2 - x}{2r} \\ &= (r + \frac{x}{2r})/2. \end{aligned}$$

Our second algorithm is based on the Newton formula.

### Code

```
class Solution {
public:
    int mySqrt(int x) {
        if(x==0) return 0;
```

```

        int u=x, l=1;
        int mid=l+(u-l)/2;
        while(mid!=u && mid!=l){
            if(x/mid<mid) u=mid;
            else l=mid;
            mid=l+(u-l)/2;
        }
        return l;
    }
};

```

Two ways to avoid overflow in the codes are to use `mid=l+(u-l)/2;` and `x/mid<mid` instead of `mid=(l+u)/2;` and `mid*mid>x`.

### Code\*: Newton's method

```

class Solution {
public:
    int mySqrt(int x) {
        long long r = x;
        while (r*r > x)
            r = (r + x/r) / 2;
        return r;
    }
};

```





# Appendix A

## Chapter Heading

*All's well that ends well*

Use the template *appendix.tex* together with the Springer document class *SVMono* (monograph-type books) or *SVMult* (edited books) to style appendix of your book in the Springer layout.

### A.1 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the  $\LaTeX$  automatism for all your cross-references and citations.

#### A.1.1 Subsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the  $\LaTeX$  automatism for all your cross-references and citations as has already been described in Sect. A.1.

For multiline equations we recommend to use the `eqnarray` environment.

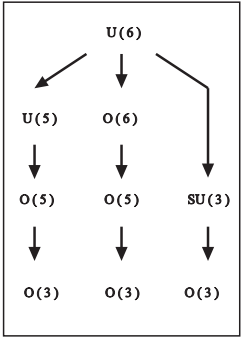
$$\begin{array}{l} \mathbf{a} \times \mathbf{b} = \mathbf{c} \\ \mathbf{a} \times \mathbf{b} = \mathbf{c} \end{array} \quad (\text{A.1})$$

#### Subsubsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the  $\LaTeX$  automatism for all your cross-references and citations as has already been described in Sect. A.1.1.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

**Fig. A.1** Please write your figure caption here



**Table A.1** Please write your table caption here

Classes	Subclass	Length	Action Mechanism
Translation	mRNA <sup>a</sup>	22 (19–25)	Translation repression, mRNA cleavage
Translation	mRNA cleavage	21	mRNA cleavage
Translation	mRNA	21–22	mRNA cleavage
Translation	mRNA	24–26	Histone and DNA Modification

<sup>a</sup> Table foot note (with superscript)

# Glossary

Use the template *glossary.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your glossary in the Springer layout.

**glossary term** Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

**glossary term** Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

**glossary term** Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

**glossary term** Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

**glossary term** Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.