课程目标：

让大家在面对面试中的算法问题时，有一个合理思考路径；

面对算法面试，不畏惧。因为面试中的算法问题，通常并不复杂，远远不需要啃完一本《算法导论》

第一章、算法面试到底是什么鬼？

1-1 算法面试不仅仅是正确的回答问题

算法面试是什么？

答：不代表能够正确回答每一个算法问题，但是合理的思考方向其实更重要，也是完成算法面试问题的前提。

算法面试优秀并不意味着技术面试优秀，技术面试优秀并不意味着能够拿到 offer。

什么是给出一个合理的思考路径？

答：算法面试的目的不是给出一个“正确”答案，而是向面试官展示你思考问题的方式。

1-2 算法面试只是面试的一部分

自己简历中的项目中梳理一下技术要点。

你遇到的印象最深的 bug 是什么？

面向对象

设计模式

网络相关、安全相关、内存相关、并发相关。

遇到的最大的挑战？

犯过的错误？

遭遇的失败？

最享受的工作内容？

遇到冲突的处理方式？

做的最与众不同的事儿？

准备好合适的问题文面试官？

注意：面试官问这些问题一定要用结合具体的项目来回答。

准备好合适的问题问面试官，比如：

整个小组的大概运行模式是怎样的？

整个项目后续规划是如何的？

这个产品的某个问题是如何解决的？

为什么会选择某些技术？标准？

我对某个技术很感兴趣，在你的小组中我会有怎样的机会深入了解这项技术。

1-3 如何准备算法面试

1.远远不需要啃完一本《算法导论》，因为它强调理论证明。

2.高级数据结构和算法面试提及概率很低。（了解，不需要实现）

比如：红黑树、B-Tree、斐波那契堆、计算几何、数论、FFT

3.不要轻视基础算法和数据结构，而只关注“有意思”的题目。

--各种排序算法

--基础数据结构和算法的实现：如堆、二叉树、图

--基础数据结构的使用：如链表、栈、队列、哈希表、图、Trie、并查集...

--基础算法：深度优先、广度优先、二分查找、递归...

--基本算法思想：递归、分治、回溯、贪心、动态规划。

1-4 如何回答算法面试问题

第二章、面试中的复杂度分析

2-1 究竟什么是大 O（Big O）

2.2 对数据规模有个概念

如果想在 1s 之内解决问题：

O(n^2)可以处理大约 10^4

O(n)可以处理大约 10^8

O(nlogn)可以处理大约 10^7

2.3 简单复杂度分析

O(1):常数

```
void swapTwoInts(int &a,int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

O(n):线性

```
int reverse(string &s)
{
    int n = s.size();
    for(int i = 0;i < n/2;i++)
        swap(s[i],s[n-1-i]);
}
```

O(n^2)

```
void selectionSort(int arr[],int n)
{
    for(int i = 0; i < n; i++)
    {
        int minIndex = i;
        for(int j = i + 1; j < n; j++)
            if(arr[j] < arr[minIndex])
                minIndex = j;
        swap(arr[i],arr[minIndex])
    }
}
```

(n-1)+(n-2)+...+1+0 = n(n-1)/2=1/2*n^2-1/2*n=O(n^2)

O(logn)

```
int binarySearch(int arr[],int n,int target)
{
    int l = 0; r = n - 1;
    while( l <= r)
    {
        int mid = l + (r - l) / 2;
        if(arr[mid] == target) return mid;
        if(arr[mid] > target) r = mid - 1;
        else l = mid + 1;
    }
    return -1;
}
```

O(nlogn)

```
void hello(int n)
{
    for(int sz = 1;sz < n;sz += sz)
        for(int i = 1;i < n;i++)
            cout << "hello" << endl;
}
```

O(sqrt(n))

```
bool isPrime(int n)
{
    for(int x = 2;x*x <= n;x++)
        if(x%x == 0)
            return false
    return true;
}
```

2.5 递归算法的复杂度分析

如果递归函数中，只进行一次递归调用，递归深度为 depth;

在每个递归函数中，时间复杂度为 T;则总体的时间复杂度为 O(T*depth)

```
int sum (int n)
{
    assert(n >= 0);
    if(n == 0)
        return 0;
    return n + sum(n - 1);
}
```

递归深度:n

时间复杂度:O(1)

总体时间复杂度：O(n)

递归中进行多次递归调用，复杂度是调用次数

```
int f(int n)
{
    assert(n >= 0);
    if(n == 0)
        return 1;
    return f(n-1) + f(n - 1);
}
```

办法：画出递归树

则：1+2+4+8+...

=2^0+2^1+2^2+...+2^n

=2^(n-1)-1

=O(2^n)

2.6 均摊时间复杂度分析：vector 的 resize()需要均摊到每个 push_back()操作

3. 数组中的问题其实很常见

3.1 从二分查找法看如何写出正确的程序

数组中的问题其实最常见

排序：选择排序；插入排序；归并排序；快速排序

查找：二分查找

出具结构：栈；队列；堆

二分查找法

二分查找法的思想在 1946 年提出，但是第一个没有 bug 的二分查找法在 1962 年出现。

对于有序的数列，才能使用二分查找法（排序的作用）

```
template<typename T>
int binarySearch(T arr[],int n,T target)
{
    int l=0,r=n-1;  //要清楚 l 和 r 的实际意义，如在[l...r]的范围里寻找 target
    while( l <= r ){ //当 l == r 时，区间依然有元素，所以还应该查找下去
        int mid = (l+r)/2;    //整形溢出 bug
        int mid = l + (r-l)/2;
        if(arr[mid] == target)
            return mid;
        if(target > arr[mid])
            l = mid + 1;     //target 在[mid+1...r]中
        else
            r = mid - 1;     //target 在[l...mid-1]中
```

```
        }
        return -1;
    }
```

3.2 改变变量定义，依然可以写出正确的算法

一定要清楚变量代表什么意义。

1.明确变量的含义

2.循环不变量

3.小数据量调试

4.大数据量测试

3.3 在 leetcode 上解决第一个问题 Movw Zeros

给定一个数组 nums，写一个函数，将数组中所有的 0 挪到数组的尾部，而维持其他所有非 0 元素的相对位置。

举例：nums=[0,1,0,3,12]，函数运行后结果为[1,3,12,0,0]

思路 1：把非 0 保存出来，填在 nums 左端，后端补 0

```
void moveZeros(vector<int>& nums)
{
    时间复杂度 O(n)   空间复杂度 O(n)
    vector<int> nonZeroElements;
    for(int i = 0;i < nums.size();i++)
        if(nums[i])
            nonZeroElements.push_back(num[i]);
    for(int i = 0;i < nonZeroElements.size();i++)
        nums[i] = nonZeroElements[i];
    for(int i = nonZeroElements.size();i<nums.size();i++)
        nums[i] = 0;
}
```

思路 2:2 个指针，原地

```
void moveZeros(vector<int>& nums)
{
    int k = 0;  //nums 中，[0...k)的元素均为非 0 元素
    //遍历到第 i 个元素后，保证[0,...k)中所有非 0 元素都按照顺序排列在[0...k)中
    for(int i = 0;i < nums.size();i++)
        if(nums[i])
            nums[k++] = nums[i];
    //将 nums 剩余位置放置为 0
    for(int i = k;i < nums.size();i++)
        nums[i] = 0;
}
```

思路 3：不是赋值，用交换可以避免对后序填 0

```
void moveZeros(vector<int>& nums)
{
```

```
    int k = 0;  //nums 中，[0...k)的元素均为非 0 元素
    //遍历到第 i 个元素后，保证[0,...i]中所有非 0 元素都按照顺序排列在[0...k)中,同时，[k...i]为 0
    for(int i = 0;i < nums.size();i++)
        if(nums[i])
            if( i != k)
                swap(nums[k++] = nums[i]);
            else
                k++;
}
```

问题 2：Remove Element

给定一个数组 nums 和一个数值 val，将数组中所有等于 val 的元素删除，并返回剩余元素个数。

如 nums=[3,2,2,3],val=3;返回 2，且 nums 中前两个元素为 2

--如何定义删除？从数组中去除？还是放在数组尾部？

--剩余元素的排列是否要包拯元游的相对顺序？

--是否有空间复杂度的要求？O(1)

问题 3:Remove Duplicated from Sorted Array

给定一个有序数组，对数组中的元素去重，是的原数组的每个元素只有一个。返回去重后数组的长度值

如 nums=[1,1,2],结果颖返回 2，且 nums 的前两个元素为[1,2]

--如何定义删除？从数组中去除？还是放在数组尾部？

--剩余元素的排列是否要包拯元游的相对顺序？

--是否有空间复杂度的要求？O(1)

问题 3:Remove Duplicated from Sorted Array II

给定一个有序数组，对数组中的元素去重，是的原数组的每个元素只有一个。返回去重后数组的长度值

如 nums=[1,1,1,2,2,3],结果颖返回 5，且 nums 的前两个元素为[1,1,2,2,3]

--如何定义删除？从数组中去除？还是放在数组尾部？

--剩余元素的排列是否要包拯元游的相对顺序？

--是否有空间复杂度的要求？O(1)

3.5 三路快排 partition 思路的应用 Sort Color

基础算法思路的应用

Sort Colors

给定一个有 n 个元素的数组，数组中元素的取值只有 0,1,2 三种可能。为这个数组排序。

-可以使用任意一种排序算法

-没有使用上题目中给出的特殊条件

计数排序：分别统计 0,1,2 的元素个数，然后放回原有数组

```
void sortColors(vector<int>& nums)
{
    //时间复杂度:O(n)
    //空间复杂度:O(k),k=3,即 O(1)
    int count[3] = {0};
```

```
        for(int i = 0;i < nums.size();i++)
        {
            assert(nums[i] >= 0 && nums[i] <= 2);
            count[nums[i]]++;
        }
        int index = 0;
        for(int i = 0; i < count[0]; i++)
            nums[index++] = 0;
        for(int i = 0; i < count[1]; i++)
            nums[index++] = 1;
        for(int i = 0; i < count[2]; i++)
            nums[index++] = 2;
    }
```

三路快排思想

```
    void sortColors(vector<int>& nums)
    {
        //只遍历了数组一变
        int zero = -1;   //nums[0...zero] = 0,所以 zero = -1******注意
        int two = nums.size();          //nums[two...n-1] = 2,所以 two = nums.size()******注意
        for(int i = 0; i < two; )
        {
            if(nums[i] == 1)
                i++;
            else if(nums[i] == 2)
                swap(nums[i],nums[--two]);
            else
            {
                assert(nums[i] == 0);
                swap(nums[++zero],nums[i++]);
            }
        }
    }
```

问题：Merge Sorted Array

给定两个有序整形数组 nums1,nums2，将 nums2 的元素归并到 num1 中

问题：Kth Largest Element in an Array

在一个整数序列中寻找第 k 大的元素

--如给定数组[3,2,1,5,6,4],k=2，结果为 5

利用快排 partition 中，将 pivot 放置在了其正确的位置上的性质

3.6 对撞指针 Two Sum II

给定一个有序整形数组和一个整数 target，在其中寻找两个元素，使得其和为 target。返回两个数的索引

--如 numbers=[2,7,11,15],target=9

--返回数字 2,7 的索引 1,2(索引从 1 开始计算)

要跟面试官确认

--如果没有解怎样？保证有解

--如果有多个解怎样？返回任意解

1.最直接的思考：暴力解法。双层遍历，O(n^2)

暴力解法没有充分利用源数组的性质---有序

2.看到有序，首先想到二分搜索

O(nlogn)二分解法

3.对撞指针法:时间复杂度 O(n)，空间复杂度 O(1)

```
vector<int> twoSum(vector<int>& numbers, int target)
{
    assert(numbers.size() >= 2);
    int l = 0,r = numbers.size() - 1;
    while(l < r)
    {
        if(numbers[l] + numbers[r] == target)
        {
            int res[2] = {l+1,r+1};
            return vector<int>(res,res+2);
        }
        else if(numbers[l] + numbers[r] < target)
            l++;
        else
            r--;
    }
    throw invalid_argument("The input has no solution.");
}
```

问题 2：Valid Palindrome

给定一个字符串，只看其中的数字和字母，忽略大小写，判断这个字符串是否为回文串？

"A man,a plan,a canal;Panama"　是回文串

对于字符串要注意以下几个问题：

--空字符串如何看

--字符的定义

--大小写问题

问题 3：Reverse Vowels of a String

给定一个字符串，将改字符串中的元音字母翻转

--如：给出"hello"，返回"holle"

--如：给出"leetcode"，返回"leotcede"

--元音不包含 y

问题 4:Container With Most Water

给出一个非负整数 a1,a2,a3,...,an;每一个整数表示一个竖立在坐标轴 x 位置的一堵高度为 ai 的“墙”，选择两堵墙，和 x 轴构成的容器可以容纳最多的水。

3.7 滑动窗口 Minimun Size Subarray Sum

双索引技术 Two Pointer

问题 1：Minimun Size Subarray Sum

给定一个整形数组和一个数字 s，找到数组中最短的一个连续子数组，使得连续子数组和 sum>=s，返回这个最短的连续子数组的返回值

--如，给定数组[2,3,1,2,4,3],s=7

--答案为[4,3]，返回 2

确认：

--什么叫子数组

--如果没有解怎么办？返回 0

--多个解怎么返回？

思路 1：暴力解：遍历所有的连续子数组[i,j],计算其和 sum,验证 sum>=s,时间复杂度 O(n^3)

思路 2：滑动窗口

```
int minSubArrayLen(int s, vector<int>& nums)
{
    时间复杂度:O(n)
    空间复杂度:O(1)
    int l=0 , r=-1;  //nums[l...r]为我们的滑动窗口
    int sum = 0;
    int res = nums.size() + 1;
    while( l < nums.size() )
    {
        if( r+1 < nums.size() && sum < s )
            sum += nums[++r];
        else
            sum -= nums[l++];
        if(sum >= s)
            res = min(res,r-l+1);
    }
    if (res == nums.size() + 1)
        return 0;
    return res;
}
```

3.8 在滑动窗口中做记录

问题 1：Longest Substring Without Repeating Characters

在一个字符串中寻找没有重复字母的最长子串

--如"abcabcbb"，则结果为"abc"

--如"bbbbb"，则结果为"b"

--如"pwwkew"，则结果为"wke"

注意：

--字符集？只有字幕？数字+字母？ASCII？

--大小写是否敏感

```
int lengthOfLongestSubstring(string s)
{
    int freq[256] = {0};
    int l = 0; r = -1; //滑动窗口为 s[l...r]
    int res = 0;
    while( l < s.size() )
    {
        if( r+1 < s.size() && freq[s[r+1]] == 0)
            freq[s[++r]]++;
        else
            freq[s[l++]]--;
        res = max(res , r-l+1);
    }
    return res;
}
```

问题 2：Find All Anagrams in a String

问题 3：Minimum Window Substring

四、查找问题

两类查找问题

查找有无

--元素'a'是否存在?set;集合

查找对应关系(键值对应)

--元素'a'出现了几次？map;字典

set 和 map

通常语言的标准库中都内置 set 和 map

--容器类

--屏蔽实现细节

--了解语言中标准库里常见容器类的使用

常见操作：

--insert

--find

--erase

--change(map)

4. 查找表的相关问题

4.1 set 的使用

问题 1：Intersection of Two Arrays

给定两个数组 nums,求两个数组的公共元素。

--如 nums1 = [1,2,2,1],nums2 = [2,2]

--结果为[2]

--结果中每个元素只能出现一次

--出现的顺序可以是任意的

```
vector<int> intersection(vector<int>& nums1, vector<int>& nums2)
{
    set<int> record(nums1.begin(), nums1.end());
    set<int> resultSet;
    for(int i = 0;i < nums2.size(); i++)
        if(record.find(nums2[i]) != record.end())
            resultSet.insert(nums2[i]);
    return vector<int>(resultSet.begin(),resultSet.end());
}
```

4.2 map 的使用

问题 1：Intersection of Two Arrays II

给定两个数组 nums，求两个数组的交集.

--如 nums1 = [1,2,2,1],nums2 = [2,2]

--结果为[2,2]

--出现的顺序可以是任意的

```
vector<int> intersection(vector<int>& nums1, vector<int>& nums2)
{
    map<int,int> record;
    for(int i = 0; i < nums1.size(); i++)
        record[nums1[i]]++;

    vector<int> resultVector;
    for(int i = 0;i < nums2.size(); i++)
        if(record.find(nums2[i]) != record.end() && record[nums2[i]] > 0)
        {
            resultVector.push_back(nums2[i]);
            record[nums2[i]]--;
        }
    result resultVector;
}
```

4.3 set 和 map 不同底层实现的区别

解决查找问题的一个很好的数据结构：hash 表

哈希表的缺点是失去了数据的顺序性

map 和 set 的底层实现为平衡二叉树

unordered_map 和 unoedered_set 的底层实现为哈希表

问题 1：Valid Anagram

问题 2：Happy Number

问题 3：Word Pattern

问题 4：Isomorphic Strings

问题 5：Sort Characters By Frequency

4.4 使用查找表的经典问题 Two Sum

问题 1：Two Sum

给出一个整形数组。返回这个数组中的两个数字的索引值 i 和 j，使得 nums[i]+nums[j]等于一个给定的 target 值。两个索引不能相等。

--如 nums = [2,7,11,15],target = 9

--返回[0,1]

考虑：

--索引从 0 开始计算还是从 1 开始计算？

--没有解怎么办？

--有多个解怎么办？保证有唯一解。

思路 1：暴力解法 O(n^2)

思路 2：排序后，使用双索引对撞 O(nlogn)+O(n)=O(nlogn)

思路 3：查找表（这是个查找问题）。将所有元素放入查找表，之后对于每一个元素 a，查找 target-a 是否存在。

```
vector<int> twoSum(vector<int>& nums, int target)
{
    //时间复杂度:O(n)
    //空间复杂度:O(n)
    unorder_map<int,int> record;
    for(int i = 0; i < nums.size(); i++)
    {
        int complement = target - nums[i];
        if(record.find(complement) != record.end())
        {
            int res[2] = {i,record[complement]}
            return vector<int>(res,res+2);
        }
        record[nums[i]] = i;
    }
    throw invalid_argument("the input has no solution");
}
```

问题 2：3Sum

给出一个整形数组，寻找其中的所有不同的三元组(a,b,c)，使得 a+b+c=0. 如 nums=[-1,0,1,2,-1,-4]，结果为 [[-1,0,1],[-1,-1,2]]

问题 3：4Sum

给出一个整形数组，寻找其中的所有不同的四元组(a,b,c,d)，使得 a+b+c+d=0.

问题 4：3Sum Closest

给出一个整形数组，寻找其中的所有不同的三元组(a,b,c)，使得 a+b+c 的值最接近另一个给定的数字 target

--如 nums = [-1,2,1,-4],target = 1

4.5 灵活选择键值 4Sum II

给出四个整形数组 A,B,C,D,寻找有多少个 i,j,k,l 的组合，使得 A[i]+B[j]+C[k]+D[l]==0.其中，A,B,C,D 中均含有相同的元素

个数 N，且 0<=N<=500.

思路 1：暴力解发:O(n^4)

思路 2：将 D 中的元素放入查找表：O(n^3)

思路 3：将 C+D 的每一种可能放入查找表：O(n^2)

```
int fourSumCount(vector<int>& A,vector<int>& B,vector<int>& C,vector<int>& D)
{
    //时间复杂度 O(n^2)
    //空间复杂度 O(n^2)
    assert(A.size() == B.size() && B.size() == C.size() && C.size() == D.size());
    unordered_map<int,int> record;
    for(int i = 0; i < C.size(); i++)
        for(int j = 0; j < D.size();j++)
            record[C[i]+D[j]]++;
    int res = 0;
    for(int i = 0;i < A.size(); i++)
        for(int j = 0; j < B.size(); j++)
            if(record.find(0-A[i]-B[j]) != record.end())
                res += record[0-A[i]-B[j]];
    return res;
}
```

问题 2：Group Anagrams

4.6 灵活选择键值 Number of Boomerangs

给出一个平面上的 n 个点，寻找存在多少个由这些点构成的三元组(i,j,k)，使得 i,j 两点的距离等于 i,k 两点的距离。其中 n 最多为 500，且所有的点坐标的范围在[-10000,10000]之间。

--如[[0,0],[1,0],[2,0]]，则结果为 2

--两个结果为[[1,0],[0,0],[2,0]]和[[1,0],[2,0],[0,0]]

思路 1：暴力解法:O(n^3)

思路 2：map 中存距 i 点的所有距离及点数

```
int numberOfBoomerangs(vector<pair<int,int>>& points)
{
    //时间复杂度 O(n^2)
    //空间复杂度 O(n)
```

```
        int res = 0;
        for(int i = 0; i < points.size(); i++)
        {
            unordered_map<int,int> record;
            for(int j = 0;j < points.size();j++)
                if(j != i)
                    record[dis(points[i],points[j])]++;
            for(unordered_map<int,int>::iterator iter = record.begin();iter != record.end(); iter++)
            {
                if(iter->second >= 2)
                    res += (iter->second)*(iter->second-1);
            }
        }
        return res;
    }
    int dis(const pair<int,int>& pa,const pair<int,int>& pb)
    {
        return (pa.first-pb.first)*(pa.first-pb.first)+(pa.second-pb.second)*(pa.second-pb.second);
    }
```

问题 2：Max Points on a Line

4.7 查找表和滑动窗口

问题 1：Contains Dumplicate II

给出一个整形数组 nums 和一个整数 k，是否存在索引 i 和 j，使得 nums[i]==nums[j]且 i 和 j 之间的差不超过 k

思路 1：暴力解法:O(n^2)

思路 2：滑动窗口

```
    bool containsNearbyDuplicate(vector<int>& nums, int k)
    {
        //时间复杂度:O(n)
        //空间复杂度:O(k)
        unordered_set<int> record;
        for(int i = 0;i < nums.size();i++)
        {
            if(record.find(nums[i]) != record.end())
                return true;
            record.insert(nums[i]);
            //保持 record 中最多有 k 个元素
            if(record.size() == k + 1)
                record.erase(nums[i-k]);
        }
        return false;
```

```
}
```

问题 2:Contains Duplicate

4.8 二分搜索树底层实现的顺序性 Contain Duplicate III

给出一个整形数组 nums，是否存在索引 i 和 j，使得 nums[i]和 nums[j]之间的差别不超过给定的整数 t,且 i 和 j 之间的差别不超过给定的整数 k

思路 1：滑动窗口，在 l+1...l+k 之间，寻找 fabs(v-x)<=t 的元素，其中 v 为第 l+k+1 的元素

```
bool containsNearbyAlmostDuplicate(vector<int>& nums, int k,int t)
{
    //时间复杂度:O(n)
    //空间复杂度:O(k)
    set<int> record;
    for(int i = 0;i < nums.size();i++)
    {
        if(record.lower_bound(nums[i]-t) != record.end() && *record.lower_bound(nums[i]-t) <= nums[i]+t)
            return true;
        record.insert(nums[i]);
        //保持 record 中最多有 k 个元素
        if(record.size() == k + 1)
            record.erase(nums[i-k]);
    }
    return false;
}
```

5. 在链表中穿针引线

5.1 链表

问题 1：在节点间穿针引线 Reverse Linked List

需要三个指针(pre,cur,next)

```
struct ListNode{
    int val;
    ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};
ListNode* reverseList(ListNode* head)
{
    时间复杂度 O(n)
    空间复杂度 O(1)
    ListNode* pre = NULL;
    ListNode* cur = head;
    while(cur != NULL)
    {
        ListNode* next = cur->next;
```

```
            cur->next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
```

问题 2：Reverse Linked List II

翻转一个链表从 m 到 n 的元素

如对于链表 1->2->3->4->5->NULL,m=2,n=4

则返回链表 1->4->3->2->5->NULL

--m 和 n 超过链表范围怎么办？

--m>n 怎么办？

5.2 测试你的链表程序

```
    ListNode* createLinkedList(int arr[],int n)
    {
        if(n == 0)
            return NULL;
        ListNode* head = new ListNode(arr[0]);

        ListNode* curNode = head;
        for(int i = 1;i < n; i++)
        {
            curNode->next = new ListNode(arr[i]);
            curNode = curNode->next;
        }
        return head;
    }
    void deleteLinkedList(ListNode* head)
    {
        ListNode* curNode = head;
        while(curNode != NULL)
        {
            ListNode* delNode = curNode;
            curNode = curNode->next;
            delete delNode;
        }
        return;
    }
    void printLinkedList(ListNode* head)
    {
```

```
    ListNode* curNode = head;
    while(curNode != NULL)
    {
        cout << curNode->val << " -> ";
        curNode = curNode->next;
    }
    cout << "NULL" << endl;
}
```

问题 2.Remove Duplicates from Sorted List

问题 3.Partition List

问题 4.Odd Even Linked List

问题 5.Add Two Numbers

问题 6.Add Two Numbers II

5.3 设立链表的虚拟头结点

问题 1:Remove Linked List Elements

在链表中删除值为 val 的所有节点

```
--如 1->2->6->3->4->5->6->NULL
--返回 1->2->3->4->5->NULL
ListNode* removeElements(ListNode* head,int val)
{
    if(head == NULL)
        return head;
    while(head != NULL && head->val == val){
        ListNode* delNode = head;
        head = delNode->next;
        delete delNode;
    }
    if(head == NULL)
        return NULL;
    ListNode* cur = head;
    while(cur->next != NULL)
    {
        if(cur->next->val == val)
        {
            ListNode* delNode = cur-next;
            cur->next = delNode->next;
            delete delNode;
        }
        else
            cur = cur->next;
```

```
        }
        return head;
    }
```

虚拟头结点：

```
ListNode* removeElements(ListNode* head,int val)
{
    ListNode* dummyHead = new ListNode(0);
    dummyHead->next = head;

    ListNode* cur = dummyHead;
    while(cur->next != NULL)
    {
        if(cur->next->val == val)
        {
            ListNode* delNode = cur-next;
            cur->next = delNode->next;
            delete delNode;
        }
        else
            cur = cur->next;
    }
    ListNode* retNode = dummyHead->next;
    delete dummyHead;
    return retNode;
}
```

问题 2：Remove Dumplicates from Sorted List II

问题 3：Merge Two Sorted Lists

5.4 复杂的穿针引线

问题 1：Swap Nodes in Pairs

给定一个链表，对于每两个相邻的节点，交换其位置。

-如：链表为 1->2->3->4->NULL

-返回:2->1->4->3->NULL

-只能对节点进行操作，不能修改节点的值

```
ListNode* swapPairs(ListNode* head)
{
    ListNode* dummyHead = new ListNode(0);
    dummyHead->next = head;

    ListNode* p = dummyHead;
    while(p->next && p->next->next)
```

```
        {
            ListNode* node1 = p->next;
            ListNode* node2 = node1->next;
            ListNode* next = node->next;

            node2->next = node1;
            node1->next = next;
            p->next = node2;
        }
        ListNode* retNode = dummyHead->next;
        delete dummyHead;
        return retNode;
    }
```

问题 2：Reverse Nodes in k-Group

问题 3：Insertion Sort List

问题 4；Sort List,写一个排序算法，用 O(nlogn)的时间复杂度为一个链表进行排序（归并排序）

5.5 不仅仅是穿针引线

问题 1：Delete Node in a Linked List

给定链表中的一个节点，删除该节点

```
void deleteNode(ListNode* node)
{
    if(node == NULL)
        return;
    if(node->next == NULL)
    {
        delete node;
        node = NULL;
    }
    node->val = node->next->val;
    ListNode* delNode = node->next;
    node->next = delNode->next;
    delete delNode;
    return;
}
```

5.6 链表和双指针

问题 1：Remove Nth Node Form End of List

给定一个链表，删除倒数第 n 个节点

-如：1->2->3->4->5->NULL,n=2

-返回:1->2->3->5

```
class Solution {
```

```
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummyNode = new ListNode(0);
        dummyNode->next = head;

        ListNode* p = dummyNode;
        ListNode* q = dummyNode;
        for(int i = 0;i < n;i++)
            q = q->next;
        while(q->next)
        {
            p = p->next;
            q = q->next;
        }
        ListNode* delNode = p->next;
        p->next = delNode->next;
        delete delNode;

        ListNode* retNode = dummyNode->next;
        delete dummyNode;
        return retNode;
    }
};
```

问题 2：Rotate List

问题 3：Reorder List

问题 4：Palindrome Linked

6. 栈，队列，优先队列

6.1 栈的队列的基础应用

问题 1：Valid Parenttheses

给定一个字符串，只包含(,[,{,),],},判断字符串中的括号匹配是否合法。

--如"()"，"()[]{}"是合法的

--如"(]"，"([)]"是非法的

```
bool isValid(string s)
{
    stack<char> stack;
    for(int i = 0;i < s.size(); i++)
    {
        if(s[i] == '(' || s[i] == '{' || s[i] == '['
            stack.push(s[i]);
        else
```

```
            {
                if(stack.size() == 0)
                    return false;
                char c = stack.top();
                stack.pop();

                char match;
                if(s[i] == ')')
                    match = '(';
                else if(s[i] == ']')
                    match = '[';
                else
                {
                    assert(s[i] == '}');
                    match = '{';
                }
                if(c != match)
                    return false;
            }
        }
        if(stack.size() == 0)
            return false;
        return true;
    }
```

问题 2：Evaluate Reverse Polish Notation

问题 3：Simplify Path

6.2 栈和递归的紧密关系

问题 1：Binary Tree Preorder Traversal

问题 2：Binary Tree Inorder Traversal

问题 3：Binary Tree Postorder Traversal

利用栈实现递归

6.3 运用栈模拟递归

问题 1：Binary Tree Preorder Traversal

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

```
        string s;
        TreeNode* node;
        Command(string s,TreeNode* node):s(s),node(node){}
    };
    vector<int> preorderTraversal(TreeNode* root)
    {
        vector<int> res;
        if(root == NULL)
            return res;
        stack<Command> stack;
        stack.push(Command("go",root));
        while(!stack.empty()){
            Command command = satck.top();
            stack.pop();
            if(command.s == "print")
                res.push_back(command.node->val);
            else
            {
                assert(command.s == "go");
                if(command.node->right)
                    stack.push(Command("go",command.node->right));
                if(command.node->left)
                    stack.push(Command("go",command.node->left));
                stack.push(Command("print",command.node));
            }
        }
        return res;
    }
```

问题 2：Flatten Nested List Lterator

6.4 队列的典型应用

问题 1：Binary Tree Level Order Traversal

对二叉树进行层序遍历

```
    vector<vector<int>> levelOrder(TreeNode* root)
    {
        vector<vector<int>> res;
        if(root == NULL)
            return res;
        queue< pair<TreeNode*,int> > q;
        q.push(make_pair(root,0));
        while(!q.empty())
```

```
        {
            TreeNode* node = q.front().first;
            int level = q.front().second;
            q.pop();

            if(level == res.size())
                res.push_back(vector<int>());
            res[level].push_back(node->val);
            if(node->left)
                q.push(make_pair(node->left,level+1));
            if(node->right)
                q.push(make_pair(node->right,level+1));
        }
    }
```

问题 2：Binary Tree Level Order TravelsalII

问题 3：Binary Tree Zigzag Level Order Traversal

问题 4：Binary Tree Right Side View

6.5 BFS 和图的最短路径

问题 1：Perfect Squares

给出一个正整数，寻找最少的完全平方数，使他们的和为 n

--完全平方数:1,4,9,16

--12 = 4 + 4 + 4

--13 = 4 + 9

6.6 优先队列

优先队列的底层实现：堆

使用优先队列解决算法问题

C++ : priority_queue

priority_queue<int> pq1:默认最大堆

priority_queue<int,vector<int>,greater<int>> pq2：最小堆

```
//自定义比较
bool myCmp(int a, int b){
    return a%10 < b%10;
}
priority_queue<int,vector<int>,dunction<bool(int,int)>> pq3(myCmp);
```

6.7 优先队列相关的算法问题

问题 1：给定一个非空数组，返回前 K 个出现频率最高的元素。

--如给定[1,1,1,2,2,3]

--返回[1,2]

--注意 K 的合法性问题

思路 1：扫描一遍统计频率；排序找到前 K 个出现频率最高的元素。O(nlogn)

思路 2：维护一个含有 k 个元素的优先队列。如果遍历到的元素比队列中的最小频率元素的频率高，则去除队列中最小频率的元素，将新元素入队。最终，队列中剩下的，就是前 K 个出现频率最高的元素。O(nlogk)

```
vector<int> topKFrequent(vector<int>& nums,int k)
{
    assert(k > 0);
    //统计每个元素出现的频率
    unordered_map<int,int> freq;
    for(int i = 0;i < nums.size();i++)
        freq[nums[i]]++;
    assert(k <= freq.size());
    //扫描 freq，维护当前出现频率最高的 k 个元素(频率，元素)
    priority_queue< pair<int,int>,vector<pair<int,int>>, greater<pair<int,int>> > pq;
    for(unordered_map<int,int>::iterator iter = freq.begin();
        iter != freq.end(); iter++)
    {
        if(pq.size() == k)
        {
            if(iter->second > pq.top().first)
            {
                pq.pop();
                pq.push(make_pair(iter->second,iter->first));
            }
        }
        else
            pq.push(make_pair(iter->second,iter->first));
    }
    vector<int> res;
    while(!pq.empty())
    {
        res.push_back(pq.top().second);
        pq.pop();
    }
    return res;
}
```

问题 2：Merge k Sorted List

7. 二叉树和递归

7.1 二叉树天然的递归结构

二叉树和递归

问题 1：Maximum Depth of binary Tree

求一颗二叉树的最高深度

从根节点到叶子节点的最高深度

```
struct TreeNode{
    int val;
    TreeNode* left;
    TreeNode *right;
    TreeNode(int x) : val(x) ,left(NULL), right(NULL){}
};
int maxDepth(TreeNode* root)
{
    if(root == NULL)
        return 0;
    int leftMaxDepth = maxDepth(root->left);
    int rightMaxDepth = maxDepth(root->right);
    return max(leftMaxDepth,rightMaxDepth) + 1;
}
```

问题 2：Minimum Depth of Binary Tree

从根节点到叶子节点的最短路径长度

7.2 一个简单的二叉树问题引发的血案

问题 1：Invert Binary Tree

```
TreeNode* invertTree(TreeNode* root)
{
    if(root == NULL)
        return NULL;
    TreeNode temp = root->left;
    root->left = root->right;
    root->right = temp;
    invertTree(root->left);
    invertTree(root->right);
    return root;
}
```

问题 2：Samp Tree

问题 3：Symmetric Tree

问题 4：Count Complete Tree Nodes

问题 5：Balanced binary Tree

7.3 注意递归的终止条件

问题 1：Path Sum

给出一颗二叉树以及一个数字 sum，判断在这可二叉树上是否存在一条从根到叶子的路径，

其路径上的所有节点和为 sum

```
bool hasPathSum(TreeNode* root, int sum)
{
```

```
        if(root == NULL)
            return false;
        if(root->left == NULL && root->right == NULL)
            return sum == root->val;
        if(hasPathSum(root->left, sum-root->val))
            return true;
        if(hasPathSum(root->right,sum - root->val))
            return true;
        return false;
    }
```

问题 2：Sum of Left Leaves

7.4 定义递归问题

问题 1：Binary Tree Paths

给定一颗二叉树，返回所有表示根节点到叶子节点路径的字符串。

```
      1
    /   \
  2       3
    \
      5
```

-如上图结果为["1->2->5","1->3"]

```
vector<string> binaryTreepaths(TreeNode* root)
{
    vector<string> res;
    if(root == NULL)
        return res;
    if(root->left == NULL && root->right == NULL)
    {
        res.push_back(to_string(root->val));
        return res;
    }
    vector<string> leftString = binaryTreepaths(root->left);
    for(int i = 0;i < leftString.size();i++)
        res.push_back(to_string(root->val) + "->" leftString[i]);
    vector<string> rightString = binaryTreepaths(root->right);
    for(int i = 0;i < rightString.size();i++)
        res.push_back(to_string(root->val) + "->" rightString[i]);
    return res;
}
```

问题 2：Path Sum II

问题 3：Sum Root to Leaf Numbers

7.5 更复杂的递归逻辑

问题 1：Path Sum III

给出一颗二叉树以及一个数字 sum，判断在这可二叉树上存在多少条路径，其路径上的所有节点和为 sum

```
int pathSum(TreeNode* root, int sum)
{
    if(root == NULL)
        return 0;
    int res = findPath(root,sum);
    res += pathSum(root->left,sum);
    res += pathSum(root->right,sum);
    return res;
}
//在以 node 为根节点中，寻找包含 node 的路径，和为 sum
int findPath(TreeNode* node,int num)
{
    if(node == NULL)
        return 0;
    int res = 0;
    if(node->val == num)
        res += 1;
    res += findPath(node->left,num-node->val);
    res += findPath(node->right,num-node->val);
    return res;
}
```

7.6 二分搜索树中的问题

问题 1：Lowest Common Ancestor of a Binary Search Tree

给定一颗二分搜索树和两个节点，寻找这两个节点的最近公共祖先。

--如右图所示二分搜索树

--2 和 8 的最近公共祖先为 6

--2 和 4 的最近公共祖先为 2

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,TreeNode* q)
{
    assert( p != NULL && q != NULL);
    if(root == NULL)
        return NULL;
    if(p->val < root->val && q->val < root->val)
        return lowestCommonAncestor(root->left,p,q);
    if(p->val > root->val && q->val > root->val)
        return lowestCommonAncestor(root->right,p,q);
    return root;
```

}

问题 2：Validate Binary Search Tree

问题 3：delete Node in a BST

问题 4：Convert Sorted Array to Binary Search Tree

问题 5：Kth Smallest Element in a BST

问题 6：Lowest Common Ancestor of a Binary Tree

8. 递归和回溯

8.1 树形问题

问题 1：Letter Combinations of a Phone Number

给出一个数字字符串，返回这个数字字符串能表示的所有字母组合

```
const string letterMap[10] = {
    " ",   //0
    "",          //1
    "abc",     //2
    "def",     //3
    "ghi",     //4
    "jkl",//5
    "mno",    //6
    "pqrs",    //7
    "tuv",     //8
    "wxyz"    //9
}
vector<string> res;
void findCombination(const string &digits, int index, const string &s)
{
    if(index == digits.size())
    {
        res.push_back(s);
        return;
    }
    char c = digits[index];
    assert(c >= '0' && c <= '9' && c != '1');
    string letters = letterMap[c-'0'];
    for(int i = 0;i < letters.size();i++)
        findCombination(digits,index+1,s+letters[i]);
    return;
}
vector<string> letterCombinations(string digits)
{
    if(digits == "")
```

```
        return res;
    findCombination(digits,0,"");
    return res;
}
```

8.2 什么是回溯

回溯法是暴力解法的一个主要实现手段，对于可能性随 n 变化的问题，如上题。

问题 1：Restore IP Addresses

问题 2：Palindrome Partitioning

8.3 回溯算法的应用

排列问题

问题 1：Permutations

给定一个整形数组，其中的每个元素都各不相同，返回这个元素所有排列的可能。

-如对于[1,2,3]

-返回[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

```
vector<vector<int>> res;
vector<bool> used;

//p 中保存一个有 index 个元素的排列
//向这个排列的末尾添加第 index+1 个元素，获得一个有 index+1 个元素的排列
void generatePermutation(const vector<int>& nums, int index, vector<int>& p)
{
    if(index == nums.size())
    {
        res.push_back(p);
        return;
    }
    for(int i = 0; i < nums.size(); i++)
        if( !used[i] )
        {
            //将 nums[i]添加在 p 中
            p.push_back(nums[i]);
            used[i] = true;
            generatePermutation(nums,index+1,p);
            p.pop_back();
            used[i] = false;
        }
    return;
}
vector<vector<int>> permute(vector<int>& nums)
```

```
{
    res.clear();
    if(nums.size() == 0)
        return res;
    used = vector<bool>(nums.size(),false);
    vector<int> p;
    generatePermutation(nums,0,p);
    return res;
}
```

问题 2：Permutations II

8.4 组合问题

问题 1：Combinations

给出两个整数 n 和 k，求在 1...n 这 n 个数字中选择出 k 个数字的所有组合

-如 n=4,k=2

-结果为[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]

```
vector<vector<int>> res;

//求解 C(n,k),当前已经找到的组合存储在 C 中，需要从 start 开始搜索新的元素
void generateCombinations(int n, int k, int start, vector<int> &c)
{
    if(c.size() == k)
    {
        res.push_back(c);
        return;
    }
    for(int i = start; i <= n; i++)
    {
        c.push_back(i);
        generateCombinations(n,k,i+1,c);
        c.pop_back();
    }
    return;
}
vector<vector<int>> combine(int n, int k)
{
    res.clear();
    if(n <= 0 || k <= 0 || k > n)
        return res;
    vector<int> c;
```

```
        generateCombinations(n,k,1,c);
        return res;
    }
```

8.5 回溯法解决组合问题的优化

回溯法的剪枝

```
vector<vector<int>> res;

//求解 C(n,k),当前已经找到的组合存储在 C 中，需要从 start 开始搜索新的元素
void generateCombinations(int n, int k, int start, vector<int> &c)
{
    if(c.size() == k)
    {
        res.push_back(c);
        return;
    }
    // 还有 k-c.size()个空位，所以，[i...n]中至少要有 k-c.size()个元素
    // i 最多为 n-(k-c.size()) + 1
    for(int i = start; i <= n-(k-c.size()) + 1; i++)
    {
        c.push_back(i);
        generateCombinations(n,k,i+1,c);
        c.pop_back();
    }
    return;
}
vector<vector<int>> combine(int n, int k)
{
    res.clear();
    if(n <= 0 || k <= 0 || k > n)
        return res;
    vector<int> c;
    generateCombinations(n,k,1,c);
    return res;
}
```

问题 2：Combination Sum

问题 3：Combination Sum II

问题 4：Combination Sum III

问题 5：Subsets

问题 6：Subsets II

问题 7：Binary Watch

8.6 二维平面上使用回溯法

问题 1：Word Search

给定一个二维平面的字母和一个单词，看是否可以在这个二维平面上找到该单词。其中找到这个单词的规则是，从一个字母出发，可以横向或者纵向连接二维平面上的其他字母。同一个位置的字母只能使用一次。

```
int d[4][2] = {{-1,0},{0,1},{1,0},{0,-1}};
int m,n;
vector<vector<bool>> visited;
bool inArea(int x,int y){
    return x >= 0 && x < m && y >= 0 && y < n;
}
//从 Borad[startx][starty]开始，寻找 word[index...word.size()]
bool searchWord(const vector<vector<char>>& board,const string& word,int index,int startx,int starty)
{
    if(index == word.size() - 1)
        return board[startx][starty] == word[index];
    if(board[startx][starty] == word[index])
    {
        visited[startx][starty] = true;
        //从 startx，starty 触发，向四个方向
        for(int i = 0;i < 4; i++){
            int newX = startx + d[i][0];
            int newy = starty + d[i][1];
            if( inArea(newX,newY) && !visited[newX][newY] )
                if(searchWord(board,word,index+1,newx,newy))
                    return true;
        }
        visited[startx][starty] = false;
    }
    return false;
}
bool exist(vector<vector<char>>& board,string word)
{
    m = board.size();
    assert(m > 0);
    n = board[0].size();
    visited = vector<vector<bool>>(m,vector<bool>(n,false));
    for(int i = 0; i < board.size(); i++)
        for(int j = 0;j < board[i].size(); j++)
            searchWord();
```

}

8.7 floodfill 算法，一类经典问题

问题 1：Number of Islands

给定一个二维数组，只含有 0 和 1 两个字符。其中 1 代表陆地，0 代表水域。横向和纵向的陆地连接成岛屿，被水域分割开。问给出的地图中有多少岛屿？

如：

11110	11000
11010	11000
11000	00100
00000	00011
答案为 1	答案为 3

```
class Solution
{
private:
    int d[4][2] = {{-1,0},{0,1},{1,0},{0,-1}};
    int m,n;
    vector<vector<bool>> visited;
    bool inArea(int x,int y){
        return x >= 0 && x < m && y >= 0 && y < n;
    }
    //从 grid[x][y]位置开始，进行 floodfill
    void dfs(vector<vector<char>>& grid,int x,int y)
    {
        visited[x][y] = true;
        for(int i = 0;i < 4; i++)
        {
            int newX = x + d[i][0];
            int newY = y + d[i][1];
            if(inArea(newX,newY) && !visited[newX][newY] && grid[newX][newY])
                dfs(grid,newX,newY);
        }
        return;
    }
}
public:
    int numIslands(vector<vector<cahr>>& grid)
    {
        m = grid.size();
        if(m == 0)
```

```
            return 0;
        n = grid[0].size();
        visited = vector<vector<bool>>(m,vector<bool>(n,false));

        int res = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0;j < n; j++)
                if(grid[i][j] == '1' && !visited[i][j])
                {
                    res++;
                    dfs(grid,i,j);
                }
        return res;
    }
};
```

问题 2：Surrounded Regions

问题 3：Pacific Atlantic Water Flow

8.8 回溯法是经典人工智能的基础

问题 1：N Queens

求 n 皇后问题的所有解

n 个皇后摆放在 n*n 的棋盘格中，使得横、竖和两个对角线方向均不会同时出现两个皇后。

```
class Solution{
private:
    vector<vector<string>> res;
    vector<bool> col;
    vector<bool> dia1;
    vector<bool> dia2;

    //尝试在一个 n 皇后问题中，摆放第 index 行的皇后位置，存放在 row 中
    void putQueen(int n, int index, vector<int> &row)
    {
        if(index == n)
        {
            res.push_back(generateBoard(n,row));
            return;
        }

        for(int i = 0; i < n; i++)
        //尝试第 index 行的皇后摆放在第 i 列
            if( !col[i] && !dia[index+i] && !dia2[index-i+n-1] )
```

```
            {
                row.push_back(i);
                col[i] = true;
                dia1[index+i] = true;
                dia2[index-i+n-1] = true;
                putQueen(n,index+1,row);
                col[i] = false;
                dia1[index+i] = false;
                dia2[index-i+n-1] = false;
                row.pop_back(i);
            }
        return;
    }
    vector<string> generateBoard(int n, vector<int> &row)
    {
        assert( row.size() == n );
        vector<string> board(n,string(n,'.'));
        for(int i = 0; i < n; i++)
            board[i][row[i]] = 'Q';
        return board;
    }
public:
    vector<vector<string>> solveNQueens(int n)
    {
        res.clear();
        col = vector<bool>(n,false);
        dia1 = vector<bool>(2*n-1,false);
        dia2 = vector<bool>(2*n-1,false);

        vector<int> row;
        putQueen(n,0,row);

        return res;
    }
};
```

问题 2：N-Queens II

问题 3：Sudoku Solver

9. 动态规划基础

9.1 什么是动态规划

问题 1：斐波那契

递归法：

```
int fib(int n)
{
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

记忆化搜索--自上向下的解决问题

```
vector<int> memo;//元素初始化为-1
int fib(int n)
{
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;
    if( memo[n] == -1 )
        memo[n] = memo[n-1] + memo[n-2];
    return memo[n];
}
```

动态规划--自下向上的解决问题

```
int fib(int n)
{
    vector<int> memo(n+1,-1);

    memo[0] = 0;
    memo[1] = 1;
    for( int i = 2; i <= n ; i++ )
        memo[i] = memo[i-1] + memo[i-2];
    return memo[n];
}
```

动态规划思路：

将原问题拆解成若干子问题，同时保存子问题的答案，使得每个子问题只求解一次，最终获得原问题的答案。

递归问题-->> 重叠子问题 -->1.记忆化搜索(自顶向下)

-->> 最优子结构 -->2.动态规划(自底向上)

当一个问题拥有重叠子问题和最优子结构的时候，可以使用动态规划

9.2 第一个动态规划问题

问题 1：Climbing Stairs

有一个楼梯，总共有 n 阶台阶。每一次，可以上一个台阶，也可以上两个台阶。

问，爬上这样一个楼梯，一共有多少不同的方法？

如：n=3，方法有：[1,1,1],[1,2],[2,1],答案为 3

递归法：

```
class Solution
{
/*private:
    vector<int> memo;
private:
    int calcWays(int n)
    {
        if(n == 0 || n == 1)
            return 1;

        if(memo[n] == -1)
            memo[n] = calcWays(n-1) + calcWays(n-2);

        return memo[n];
    }*/
public:
    int climbStairs(int n)
    {
        memo = vector<int>(n+1,-1);
        return calcWays(n);
    }
    int climbStairsDTGH(int n)
    {
        vector<int> memo(n+1,-1);

        memo[0] = 1;
        memo[1] = 1;
        for(int i = 2; i <= n; i++)
            memo[i] = memo[i-1] + memo[i-2];
        return memo[n]
    }
};
```

问题 2：Triangle

问题 3：Minimum Path Sum

9.3 发现重叠子问题

问题 1：Interger Break

给定一个正整数，可以将其分割成多个数字的和，若要让这些数字的乘机最大，求分割的方法（至少要分成两个数）

算法返回这个最大的乘积。

-如 n=2，则返回 1 (2=1+1)

-如 n=10,则返回 36 (10=3+3+4)

暴力解法：回溯遍历每一个数做分割的所有可能性。O(2^n)

动态规划：最优子结构：通过求子问题的最优解，可以获得原问题的最优解。

下面是记忆搜索解决方法：

```
class Solution
{
private:
    vector<int> memo;
    int max3(int a,int b,int c)
    {
        return max(a,max(b,c));
    }
    //将 n 进行分割(至少分成 2 部分)，可以获得最大乘积
    int breakInteger(int n)
    {
        if(n == 1)
            return 1;
        if(memo[n] != -1)
            return memo;
        int res = -1;
        for( int i = 1; i < n - 1; i++ )
            //可分割成 i + (n-i)
            res = max3(res,i*(n-i),i * breakInteger(n-i));
        memo[n] = res;
        return res;
    }
public:
    int integerBreak(int n)
    {
        assert( n>=2 );
        memo = vector<int>(n+1,-1);
```

```
        return breakInteger(n);
    }
};
```

下面是动态规划方法：

```
class Solution
{
private:

    int max3(int a,int b,int c)
    {
        return max(a,max(b,c));
    }
public:
    int integerBreak(int n)
    {
        assert( n>=2 );
        // memo[i]表示将数字分割(至少分割成两部分)后得到的最大乘积
        vector<int> memo = vector<int>(n+1,-1);
        //先设计底(自底向上)
        memo[1] = 1;
        for(int i = 2; i <= n; i++)
            //求解 memo[i]
            for(int j = 1; j <= i-1; j++)
                //分割成 j , (i-j)
                memo[i] = max3(memo[i], j*(i-j) , j*memo[i-j]);
        return memo[n];
    }
};
```

问题 2：Perfect Squares

问题 3：Decode Ways

问题 4：Unique paths

问题 5：Unique Paths II

9.4 状态定义和状态转移

问题 1：House Robber

你是一个专业的小偷，打算洗劫一条街的所有房子。每个房子里都有不同价值的宝物，但是，如果你选择偷窃连续的两栋房子，就会触发报警系统。编程求出你最多可以偷窃多少的宝物？

--如[3,4,1,2],则返回 6[3,(4),1,(2)]

--如[4,3,1,2],则返回 6[(4),3,1,(2)]

注意其中对状态的定义：

考虑偷取[x...n-1]范围里的房子 (函数的定义)

根据对状态的定义，决定状态的转移：
f(0) = max{v(0)+f(2),v(1)+f(3),v(2)+f(4),...,v(n-3+f(n-1),v(n-2),v(n-1))}(状态转移方程)

```
//记忆化搜索
class Solution
{
private:
    //memo[i]  表示考虑抢劫 nums[i...n)所能获得的最大收益
    vector<int> memo;
    //考虑抢劫 nums[index...nums.size())这个范围内的所有房子
    int tryRob(vector<int>& nums, int index)
    {
        if(index >= nums.size())
            return 0;

        if(memo[index] != -1)
            return memo[index];
        int res = 0;
        for(int i = index; i < nums.size(); i++)
            res = max(res, nums[i] + tryRob(nums,i+2));

        memo[index] = res;
        return res;
    }
public:
    int rob(vector<int>& nums)
    {
        memo = vector<int>(num.size(),-1);
        return tryRob(nums,0);
    }
};

//动态规划
class Solution
{
private:
public:
    int rob(vector<int>& nums)
```

```
    {
        int n = nums.size();
        if(n == 0)
            return 0;
        //memo[i] 表示考虑抢劫 nums[i...n-1]所能获得的最大收益
        vector<int> memo(n,-1);
        //底部
        memo[n-1] = nums[n-1];

        for(int i = n - 2; i >= 0; i--)
            //求 memo[i]
            for(int j = i; j < n; j++)
                memo[i] = max( memo[i], nums[j] + (j+2 < n ? memo[j+2] : 0) );
        return memo[0];
    }
};
```

问题 1：House Robber II

问题 2：House Robber III

问题 3：Best Time to Buy and Sell Stock with Cooldown

9.5 0-1 背包问题

有一个背包，它的容量为 C(Capacity),现在有 n 种不同的物品，编号为 0...n-1，其中每一件物品的重量为 w(i), 价值为 v(i)。问可以向这个背包中放哪些物品，使得在不超过背包容量的基础上，物品的总价值最大。

暴力解法：每一件物品都可以放进背包，也可以不放进背包。O((2^n)*n)

贪心算法？优先放入平均价值最高的物品？

思路：

状态定义 F(n,C)：考虑将 n 个物品放进容量为 C 的背包，使得价值最大

状态转移 F(i,c) = F(i-1,c)(第一种情况) = v(i) + F(i-1,c-w(i))(第二种情况),区别是放不放 i 进背包

= max(F(i-1,c),v(i) + F(i-1,c-w(i)))

自顶向下，记忆化搜索：

```
class Knapsack01
{
private:
    //记忆化空间
    vector<vector<int>> memo;
    //用[0...index]的物品，填充容积为 c 的背包的最大价值
    int bestValue(const vector<int> &w,const vector<int> &v, int index, int c)
    {
        if( index < 0 || c <= 0)
            return 0;
        if(memo[index][c] != -1)
```

```
            return memo[index][c];

        int res = bestValue(w,v,index-1,c);
        if(c >= w[index])
            res = max(res,v[index] + bestValue(w,v,index-1,c-w[index]));
        memo[index][c] = res;
        return res;
    }
public:
    int knapsack01(const vector<int> &w,const vector<int> &v, int C)
    {
        memo = vector<vector<int>>(n,vector<int>(C+1,-1));
        int n = w.size();
        return bestValue(w,v,n-1,C);
    }
};
//自底向上，动态规划
class Knapsack01
{
public:
    int knapsack01(const vector<int> &w,const vector<int> &v, int C)
    {
        assert(w.size() == v.size());
        int n = w.size();
        if(n == 0)
            return 0;

        vector<vector<int>> memo(n,vector<int>(C+1,-1));

        for(int j = 0;j <= C; j++)
            memo[0][j] = (j >= w[0] ? v[0] : 0);
        for(int i = 1; i < n; i++)
            for(int j = 0; j <= C; j++)
            {
                memo[i][j] = memo[i-1][j];
                if(j >= w[i])
                    memo[i][j] = max(memo[i][j],v[i]+memo[i-1][j-w[i]]);
            }
        return memo[n-1][C];
    }
```

};

9.6 0-1 背包问题的优化和变种

时间复杂度：O(n*C)

空间复杂度：O(n*C)

优化只能在空间复杂度上考虑

更多变种：

1.完全背包问题:每个物品可以无限使用？

2.多维费用的背包问题：要考虑物品的体积和重量两个维度？

3.物品间假如更多约束

9.7 面试中的 0-1 背包问题

问题 1：Partition Equal Subset Sum

给定一个非空数组，其中所有的数字都是正整数。问是否可以将这个数组的元素分成两部分，使得每部分的数字和相等？

-如对[1,5,11,5]，可以分成[1,5,5]和[11]两部分，元素和相等，返回 true

-如对[1,2,3,5]，无法分成元素和相等的两部分，返回 false

分析：典型的背包问题，在 n 个物品中选出一定物品，填满 sum/2 的背包

F(n,C)考虑将 n 个物品填满容量为 C 的背包

F(i,c) = F(i-1,c) || F(i-1,c-w[i])

```
class Solution
{
private:
    //memo[i]:-1 表示未计算，0 不可以填充，1 可以填充
    vector<vector<int>> memo;
    //使用 nums[0...index],是否可以完全填充一个容量为 sum 的背包
    bool tryPartition(const vector<int>& nums,int index,int sum)
    {
        if(sum == 0)
            return true;

        if(sum < 0 || index < 0)
            return false;

        if(memo[index][sum] != -1)
            return memo[index][sum] == 1;

        memo[index][sum] = tryPartition(nums,index-1,sum) || tryPartition(nums,index-1,sum-nums[index]);

        return memo[index][sum];
    }
public:
```

```
    bool canPartition(vector<int>& nums)
    {
        int sum = 0;
        for(int i = 0;i < nums.size(); i++)
        {
            assert(nums[i] > 0);
            sum += nums[i];
        }
        if(sum%2 != 0)
            return false;

        memo = vector<vector<int>>(nums.size(),vector<int>(sum/2+1,-1));

        return tryPartition(nums,nums.size()-1,sum/2);
    }
};
```

动态规划

```
class Solution
{
public:
    bool canPartition(vector<int>& nums)
    {
        int sum = 0;
        for(int i = 0;i < nums.size(); i++)
        {
            assert(nums[i] > 0);
            sum += nums[i];
        }
        if(sum%2 != 0)
            return false;

        int n = nums.size();
        int C = sum/2;
        vector<bool> memo(C+1,false);

        for(int i = 0;i <= C; i++)
            memo[i] = (nums[0] == i);
        for(int i = 1;i < n; i++)
            for(int j = C; j >= nums[i]; j--)
```

```
                memo[j] = memo[j] || memo[j-nums[i]];

        return memo[C];
    }
};
```

问题 2：Coin Change

问题 3：Combination Sum IV

问题 3：Ones and Zeros

问题 4：Word Break

问题 5：Target Sum

9.8 最长上升子序列

Lonest Increasing Subsequence

给定一个整数序列，求其中的最长上升子序列的长度。

-如[10,9,2,5,3,7,101,18]，其最长上升子序列的长度为 4.

-为[2,5,7,101]

状态 LIS(i)表示以第 i 个数字为结尾的最长上升子序列的长度

表示[0...i]的范围内，选择数字 nums[i]可以获得的最长上升子序列的长度。(i 必须用)

LIS(i) = max(1+LIS(j) if nums[i] > nums[j])，其中 j<i

```
class Solution
{
public:
    int lengthOfLIS(vector<int>& nums)
    {
        if(nums.size() == 0)
            return 0;
        //memo[i]表示以 nums[i]为结尾的最长上升子序列的长度
        vector<int> memo(nums.size(),1);
        for(int i = 1;i < nums.size(); i++)
            for(int j = 0;j < i;j++)
                if(nums[j] < nums[i])
                    memo[i] = max(memo[i],1+memo[j]);
        int res = 1;
        for(int i = 0;i < nums.size();i++)
            res = max(res,memo[i]);
        return res;
    }
};
```

问题 2：Wiggle Subsequence

9.9 LCS、最短路径，求动态规划的具体解以及更多

问题 1：最长公共子序列(LCS)

给出两个字符串 S1 和 S2，求这两个字符串的最长公共子序列的长度

S1=AAACCGTAGTTATTCGTTCTAGAA

S2=CACCCCTAAGGTACCTTTGGTTC

结果为：ACCTAGTACTTTG

S1 = ABCD

S2 = AEBD

结果为：ABD

LCS(m,n):S1[0...m]和 S2[0...n]的最长公共子序列的长度

考虑：

S1[m] == S2[n]:

LCS(m,n) = 1 + LCS(m-1,n-1)

S1[m] != S2[n]:

LCS(m,n) = max(LCS(m-1,n),LCS(m,n-1))

从最后面开始计算

自己写一下递归+记忆化搜索，动态规划

问题 2：dijkstra 也是动态规划

问题 3：Logest Increasing Subsequence

问题 4：解决 Leetcode 上更多动态规划问题

10 贪心算法基础

10.1 问题 1：Assign Cookies

假设你想给小朋友们饼干。每个小朋友最多能够给一块儿饼干。每个小朋友都有一个贪心指数，成为 g(i)，g(i)表示的是这名小朋友需要的饼干大小的最小值。同时，每个饼干都有一个大小值 s(i)。如果 s(j)>=g(i)，我们将饼干 j 分给小朋友 i 后，小朋友就会很开心。给定数组 s 和 g，问如何分配饼干，能够让最多的小朋友开心。

如 g=[1,2,3],s=[1,1],结果为 1

如 g=[1,2],s=[1,2,3],结果为 2

优先使用最大的饼干给最贪心的小朋友

*****通常实现贪心算法一般都需要对数组进行排序*****

```
class Solution{
public:
    int findContentChildren(vector<int>& g,vector<int>& s){
        sort(g.begin(),g.end(),greater<int>());//从大到小排序
        sort(s.begin(),s.end(),greater<int>());//从大到小排序

        int si = 0,gi = 0;
        int res = 0;
        while(gi < g.size() && si < s.size()){
            if(s[si] >= g[gi]){
```

```
                res++;
                si++;
                gi++;
            }
            else{
                gi++;
            }
        }
        return res;
    }
};
```

问题 2：Is Subsequence

10.2 贪心算法和动态规划的关系

问题 1：Non-overlapping Intervals

给定一组区间，问最少删除多少个区间，可以让这些区间相互不重叠。

-如[[1,2],[2,3],[3,4],[1,3]],算法返回 1

-如[[1,2],[1,2],[1,2]],算法返回 2

10.3 贪心选择性质的证明

贪心选择性质：

如果无法举出反例，如何证明贪心算法的正确性？如何应用到每个例子中？

反证法：