# Shortlisted problems

5

8

28

41

46

# Index

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 |  |  | Sol1: enumeration (naive)  Sol2: sort + two pointer  Sol3: hashmap | medium |  |
| 2 |  |  | Sol: list ops + math ops | easy |  |
| 3 |  |  | Sol1: naive enumeration  Sol2: sliding window  Sol3: sliding window+prunning | medium |  |
| 4 |  |  | Sol1: merging  Sol2: divide and conquer | hard |  |
| 5 |  |  | Sol1: naive enumeration  Sol2: naive enum + prunning  Sol3: Manacher | medium | Manacher algorithm in Sol3 is hard |
| 6 |  |  | Sol: programming | easy |  |
| 7 |  |  | Sol: math ops+corner case |  |  |
| 8 |  |  | Sol: modulation and string ops | medium | string ops: find\_first\_not\_of |
| 9 |  |  | Sol: math ops | easy |  |
| 10 |  |  | Sol1: regex  Sol2: recursion  Sol3: iteration | medium | note: regex usage |
| 11 |  |  | Sol1: enumeration  Sol2: recursion->two pointer |  |  |
| 12 |  |  | Sol1: math ops  Sol2: greedy  Sol3: modulation | easy |  |
| 13 |  |  | Sol: iteration with corner case | easy |  |
| 14 |  |  | Sol: vertical search | easy |  |
| 15 |  |  | Sol: enumeration + two pointer | medium |  |
| 16 |  |  | Sol: enumeration + two pointer | medium |  |
| 17 |  |  | Sol1: DFS  Sol2: BFS | medium |  |
| 18 |  |  | Sol: enumeration + two pointer | medium |  |
| 19 |  |  | Sol: linked list + two pointer | medium |  |
| 20 |  |  | Sol: stack | easy |  |
| 21 |  |  | Sol1: list merging  Sol2: recursion | easy |  |
| 22 |  |  | Sol1: DFS + prunning  Sol2: BFS | medium |  |
| 23 |  |  | Sol1: recursive + merge  Sol2: heap | medium |  |
| 24 |  |  | Sol: linked list ops | easy |  |
| 25 |  |  | Sol: linked list ops | easy |  |
| 26 |  |  | Sol: read/write pointer | trivial |  |
| 27 |  |  | Sol: read/write pointer | trivlal |  |
| 28 |  |  | Sol1: naive enumeration  Sol2: KMP | hard |  |
| 29 |  |  | Sol: math ops + overflow | medium |  |
| 30 |  |  | Sol: sliding window | easy |  |
| 31 |  |  | Sol: next permutation | easy |  |
| 32 |  |  | Sol: stack | easy |  |
| 33 |  |  | Sol: binary search | medium |  |
| 34 |  |  | Sol: binary search | medium |  |
| 35 |  |  | Sol: binary search | medium |  |
| 36 |  |  | Sol1: naive enumeration  Sol2: bit ops | easy |  |
| 37 |  |  | Sol: recursion | medium | hard to program |
| 38 |  |  | Sol: naive iteration | trivial |  |
| 39 |  |  | Sol: recursion(combination) | easy |  |
| 40 |  |  | Sol: recursion(combination) | easy |  |
| 41 |  |  | Sol: bucket sort | hard | \*\*\* |
| 42 |  |  | Sol1: naive iteration  Sol2: two pointer | medium |  |
| 43 |  |  | Sol: number ops | medium |  |
| 44 |  |  | Sol1: recursion  Sol2: DP  Sol3: iteration | hard |  |
| 45 |  |  | Sol: DP | medium |  |
| 46 |  |  | Sol1: next\_permutation  Sol2: recursion | medium |  |
| 47 |  |  | Sol1: next\_permutation  Sol2: recursion | medium |  |
| 48 |  |  | Sol: naive scanning | easy |  |
| 49 |  |  | Sol1: sort  Sol2: hashing + sort | easy |  |
| 50 |  |  | Sol: math ops + oveflow | easy |  |

# 

# 

# 

# 1. Two sum

|  |
| --- |
| Given an array of integers, return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution. |
| Example:  Given nums = [2, 7, 11, 15], target = 9, Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1]. |

**Sol 1: enumeration**

穷举所有可能的pair, O(n2)时间

|  |
| --- |
| // solution 1: greedy scan; O(N2) time  **vector<int> twoSum(vector<int>& nums, int target)** {  for (int n = 0; n < nums.size(); ++n) {  for (int m = n+1; m < nums.size(); ++m) {  if (nums[n]+nums[m] == target) {  return vector<int>{m, n};  }  }  }  } |

**Sol 2: sorting+two pointer**

首先排序，然后利用已知的大小顺序减少搜索空间。

令 [a1 a2 … an] 为排好序的矩阵，如果a1 + a2 > target，则任意ak + an>target，因此an 被排除。问题转化为对[a1 a2 …an－1 ]求two sum。类似地，如果a1 + a2 < target，则任意a1 + ak<target，因此a1 被排除。问题转化为对[a2 …an]求two sum。

因此问题规模随着一次比较减少1，可以得到一个线性复杂度的递归算法。但因为排序时间为O(nlogn)，所以整体复杂度是O(nlogn)

|  |
| --- |
| // soluton 2a: sort + recursion  **vector<int> twoSum(vector<int>& nums, int target)** {  vector<int> idx(nums.size());  for (int n = 0; n < nums.size(); ++n) idx[n] = n;  sort(idx.begin(), idx.end(),  [&nums](int i1, int i2) {return nums[i1] < nums[i2];});  return twoSum\_recursion(nums, idx, target, 0, nums.size()-1);  }  vector<int> twoSum\_recursion(const vector<int>& nums, const vector<int>& idx, int target, int first, int last){  int sum = nums[idx[first]] + nums[idx[last]];  if (sum == target) return vector<int>({idx[first], idx[last]});  return sum > target? twoSum\_recursion(nums, idx, target, first, last-1) :  twoSum\_recursion(nums, idx, target, first+1, last);  } |

用two pointer可以更有效地实现上面的算法

|  |
| --- |
| // solution 2: sort + two pointer; O(nlogn) sort + O(n) scan  **vector<int> twoSum(vector<int>& nums, int target)** {  // sort while preserving index  vector<int> idx(nums.size());  for (int n = 0; n < nums.size(); ++n) idx[n] = n;  sort(idx.begin(), idx.end(),  [&nums](int i1, int i2) {return nums[i1] < nums[i2];});    // two pointer scanning  int i1=0, i2=nums.size()-1;  while (i1 < i2) {  int sum = nums[idx[i1]]+nums[idx[i2]];  if (sum == target) return vector<int>{idx[i1], idx[i2]};  if (sum < target) ++i1; else --i2;  }  } |

**Sol 3: hashing**

把解法1的inner loop化简为O(1)的hash map搜索。

但如果outer loop的循环对应的是index比较小的那个数，hash map的搜索范围渐渐减小，搜索代价会很大。因此考虑outer loop循环对应index比较大的那个数，hash map的搜索范围渐渐增加。

|  |
| --- |
| // solution 3: use hashmap to do O(1) retrieval; O(n) time  vector<int> twoSum(vector<int>& nums, int target) {  unordered\_map<int, int> map;  for (int n = 0; n < nums.size(); ++n) {  auto comp = map.find(target-nums[n]);  if (comp != map.end()) {  return vector<int>{comp->second, n};  }  map[nums[n]]=n;  }  } |

# 

# 

# 2. Add two numbers

|  |
| --- |
| You are given two linked lists representing two non-negative numbers. 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. |
| Example:  Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)  Output: 7 -> 0 -> 8 |

**Sol：basic linked list/math ops**

在每次循环中，计算上次的进位和当前两个数的和，余10的部分插入结果链表，多于10的话产生下一位的进位。初始时进位为0。循环在三个加数都为0（即链表已经到尾部时）结束。

|  |
| --- |
| **ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2)** {  ListNode sentinel(0);  ListNode \*tail = &sentinel;  for (int carry = 0; l1 || l2 || carry; tail = tail->next) {  int sum = carry + (l1? l1->val:0) + (l2? l2->val:0);  carry = sum/10;  tail->next = new ListNode(sum%10);  l1 = l1? l1->next : NULL;  l2 = l2? l2->next : NULL;  }  return sentinel.next;  } |

# 3. Longest Substring Without Repeating Characters

|  |
| --- |
| Given a string, find the length of the longest substring without repeating characters. |
| Example:  Given "abcabcbb", the answer is "abc", which the length is 3.  Given "bbbbb", the answer is "b", with the length of 1.  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.) |

**Sol1：enumeration**

穷举子串起始位置，然后依次读入下一个字符并检查是否在之前出现过，检查重复性可以用bit array （这里因为key只有26个，unordered\_map overkill了）。这个算法复杂度是O(n2)。

|  |
| --- |
| **int lengthOfLongestSubstring1(string s)** {  vector<bool> occ(256, false);  int maxlen = 0;  for (int first = 0; first < s.size(); ++first) {  int last;  for (last = first; last < s.size() && !occ[s[last]]; ++last)  occ[s[last]] = true;  maxlen = max(maxlen, last - first);  fill(occ.begin(), occ.end(), false);  }  return maxlen;  } |

**Sol2：sliding window**

当发现重复字符时，子串的起始位置需要推迟到上一个重复字符之后出现，因此可以跳过一些不可能的起始位置。这样避免了一部分重复检查，但每次更新子串起始位置，需要把两个起始位置之间的字符对应的bits清零。

因为每个字符被插入字典一次，并且可能被删除出字典一次，所以复杂度是O(n).

|  |
| --- |
| **int lengthOfLongestSubstring2(string s)** {  vector<bool> occ(256, false);  int maxlen = 0;  for (int first = -1, last = 0; last < s.size(); last++) {  if (occ[s[last]])  do occ[s[++first]] = false; while (first < s.size() && s[first] != s[last]);  occ[s[last]] = true;  maxlen = max(maxlen, last - first);  }  return maxlen;  } |

**Sol3：Sliding window + prunning**

可以把bitarray改为一个int数组存储每个字符上次出现的位置，这样只需要和起始位置进行比较来判断是否在子串里出现过。这个算法是O(n)时间的复杂度。

|  |
| --- |
| **int lengthOfLongestSubstring3(string s)** {  vector<int> last\_occ(256, -1);  int maxlen = 0;  for (int first = -1, last = 0; last < s.length(); last++) {  if (last\_occ[s[last]] > first) first = last\_occ[s[last]];  last\_occ[s[last]] = last;  maxlen = max(maxlen, last - first);  }  return maxlen;  } |

# 4. Median of two sorted arrays

|  |
| --- |
| 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)). |
| Note: The median of a sorted array with 2N elements is the mean of its Nth and N+1-th element. |

**Sol1: merging**

合并两个数组，并找到中位数。合并不需要全部完成，到中位数位置就可以停止。（中位数为第(m+n+1)/2）如果数组有偶数个数，则再向后看一个元素，并求平均值。

|  |
| --- |
| **inline double next(int &p1, int &p2, const vector<int>& nums1, const vector<int>& nums2)** {  if (p1 >= nums1.size()) return nums2[p2++];  if (p2 >= nums2.size()) return nums1[p1++];  if (nums1[p1] < nums2[p2]) return nums1[p1++];  else return nums2[p2++];  }  double findMedianSortedArrays1(vector<int>& nums1, vector<int>& nums2) {  int m = nums1.size(), n = nums2.size(), mid = (m+n+1)/2;  int p1=0, p2=0, val=0;  for (int k = 0; k < mid; ++k) val = next(p1, p2, nums1, nums2);  if ((m+n)%2 == 1) return val; //total number is odd  else return (val + next(p1, p2, nums1, nums2))/2.0;  } |

1

3

5

2

4

**Sol2: Devide & conquer**

中位数把输入分为等长的两段：如右图中蓝色的部分为小于等于中位数的部分，红色部分为大于等于中位数的部分。等号成立对应总元素个数为奇数的情况。对一个长度为n的数组，可能的划分有2n+1种。我们用0到2n之间的整数index它们。index k表示蓝色部分结束于第(k-1)/2个元素，红色部分开始于第k/2个元素。

如果知道第一个数组的划分则可计算第二个。假设两个数组长度为m, n，则划分k1, k2在k1+k2 = m+n时数组被平均划分成两段。但需要额外的条件保证蓝色部分的每个数都小于等于红色部分的每个数。因此我们可以使用这个条件对k1进行二分搜索。

l1

l2

r1

r2

假设l1是第一个数组的蓝色部分的最后一个元素，r2是第一个数组红色部分的第一个元素。第二个数组对应元素为l2, r2。则：

如果l1<= r2且l2 <= r1，那么所有蓝色元素小于所有红色元素。

如果l1>r2，则k1需要向左收缩；反之k1需要向右收缩。

最后，我们考虑分割位置在数组两端的情况。此时不可能再向更左或更右收缩，因此我们总是让相关的判断满足。即令l为INT\_MIN，r为INT\_MAX。

因此该算法的复杂度是O(log(min(m, n)));

|  |
| --- |
| **double findMedianSortedArrays2(vector<int>& nums1, vector<int>& nums2)** {  int m = nums1.size(), n = nums2.size();  if (m > n) return findMedianSortedArrays2(nums2, nums1);    int k1, k2, l1, l2, r1, r2;  for (int lo = 0, hi = m\*2; lo <= hi; ) {  k1 = (lo+hi)/2;  k2 = m + n - k1;  l1 = k1==0? INT\_MIN : nums1[(k1-1)/2];  l2 = k2==0? INT\_MIN : nums2[(k2-1)/2];  r1 = k1==2\*m? INT\_MAX : nums1[k1/2];  r2 = k2==2\*n? INT\_MAX : nums2[k2/2];  if (l1 > r2) hi = k1-1;  else if (l2 > r1) lo = k1+1;  else return (max(l1, l2) + min(r1, r2))/2.0;  }  return 0;  } |

# 5. Longest Palindromic Substring

|  |
| --- |
| Given a string *S*, find the longest palindromic substring in *S*. You may assume that the maximum length of *S* is 1000, and there exists one unique longest palindromic substring. |

**Sol1:naive enumeration**

遍历对称轴，并向两边扩展。长度为n的字符串共有2n-1个可能的对称轴。

|  |
| --- |
| **string longestPalindrome1(string s)** {  int max\_len = 0, first = 0, l, r;  for (int c = 1; c < 2\*s.size(); ++c) {  for (l = (c-1)/2, r = c/2; l>=0 && r < s.size() && s[l] == s[r]; --l, ++r);  if (r-l-1 > max\_len) { max\_len = r-l-1; first = l+1;}  }  return s.substr(first, max\_len);  } |

**Sol2:enumeration with prunning**

如果子串中有许多连续重复字母的话，可以进一步优化。最长回文子串的对称轴一定处在任意个连续的重复字母的中心。不然会导致回文子串短于这些连续字母。

这样可以每次扫描一段连续重复字母的左右位置，作为扩展的起始位置。

|  |
| --- |
| **string longestPalindrome(string s)** {  int max\_len = 0, first = 0, cl, cr, l, r;  for (cl = 0; cl < s.size(); cl = cr) {  for (cr = cl; cr < s.size() && s[cr] == s[cl]; cr++);  for (l = cl-1, r = cr; l>=0 && r < s.size() && s[l] == s[r]; --l, ++r);  if (r-l-1 > max\_len) { max\_len = r-l-1; first = l+1;}  }  return s.substr(first, max\_len);  } |

**Sol3:Manacher**（由于trick不容易想到，应该面试不会要求）

思想是如果已经发现一个长子串[lmax, rmax]，且当前要检查的中心c包含在这个子串中，那么可以看c的反射c'对应的最长子串。如果以c'为中心的整个子串都包含在[lmax, rmax]中，则c对应的最长子串就是c'的反射。否则，到右边缘为rmax的位置为最长。

rmax

c

c'

cmax

lb[c']

cmax-lb[c']

我们可以用[0,..., 2N-2]表示子串中心c的可能位置，一个子串的左右边缘l, r满足l+r=c.

因为内层扩展的inner forloop每次都增加rmax，因此总的复杂度是O(n).

|  |
| --- |
| **string longestPalindrome(string s)** {  vector<int> lb(s.size()\*2-1, 0);  int rmax = -1, cmax = -1, maxlen = 0, first = 0, l, r;  for (int c = 0; c < s.size()\*2-1; ++c) {  // initialize current expansion with reflection of c wrt. cmax if possible  if (c > rmax) r = (c+1)/2;  else r = min(rmax, cmax - lb[cmax - c]);    for (l = c-r; l>=0 && r<s.size() && s[r]==s[l]; --l, ++r);    // update palindrome substring of max length and rightest boundary  if (maxlen < r-l-1) {maxlen = r-l-1; first = l+1;}  if (rmax < r-1) { rmax = r-1; cmax = c;}    lb[c] = l+1;  }  return s.substr(first, maxlen);  } |

# 

# 6. Zigzag conversion

|  |
| --- |
| 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". |

**Corner case**：

需要确定nRows <=2时的行为。这里假定nRow>=0.

nRows =1时，此时按前面的公式循环长度为0，但实际上只是没有斜行，因此输出为输入本身。

nRows =2时，此时第一行为偶数index的字母，第二行为奇数index的字母。

**Sol: modularization**

这个zig zag pattern可以拆分成长度为nRows\*2-2的小段。第一行和最后一行的字母之间间隔nrow\*2-2个，首字母分别为输入string的第0个和第nRows-1个。其余行（假设行号为n)，则间隔地输入以第n个和第nRows\*2-2-n个开始的字母，循环周期仍然是nRows\*2-2.

我们需要对nRows = 1的情况特殊处理。nRows = 2的情况为中间行为空集，不用特殊处理。

|  |
| --- |
| **string convert(string s, int numRows)** {  if (numRows == 1) return s;  string str\_out("");  for (int p = 0; p < s.size(); p+=numRows\*2-2) str\_out += s[p];  for (int n = 1; n <= numRows-2; ++n) {  for (int p = n, q = numRows\*2-n-2; p < s.size(); p+=numRows\*2-2, q+=numRows\*2-2) {  str\_out +=s[p];  if (q < s.size()) str\_out+=s[q];  }  }  for (int p = numRows-1; p < s.size(); p+=numRows\*2-2) str\_out +=s[p];  return str\_out;  } |

# 

# 7. Reverse integer

|  |
| --- |
| Reverse digits of an integer. |
| Example: x = 123, return 321 |

Corner cases:

1. x < 0 : 可能的回答是返回-reverse(-x).
2. x ends with 0 : 可能的回答是返回reverse(x)的值，如reverse(100) = reverse(10) = 1
3. reverse(x) overflow: 可能的回答是返回0，或throw exception

corner case(1)(2)可以和处理大多数正整数的case合并。

**Sol 1: check overflow with long long**

|  |
| --- |
| **int reverse(int x)** {  long long result = 0;  for (; x != 0; x /= 10) result = result\*10 + x%10;  return (result < INT\_MIN || result > INT\_MAX)? 0:result;  } |

**Sol2: check overflow at hot spot**

在溢出前检查是否可能溢出(乘10后溢出，对该问题，不可能乘10后没有溢出但加了一位数后溢出)。

|  |
| --- |
| **int reverse(int x)** {  int result = 0;  for (; x != 0; x /= 10) {  if ((x > 0 && result > INT\_MAX/10) || (x < 0 && result < INT\_MIN/10)) return 0;  result = result\*10 + x%10;  }  return result;  } |

# 8. String to Integer (atoi)

|  |
| --- |
| Implement atoi to convert a string to an integer. |
| 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. |

**corner cases**

1. +号是否可以出现：（可以）
2. +-号除否可以连续出现：（只出现开始的一次，也可能不出现）
3. 是否可以让数字串以0开始：可以
4. 数字溢出怎么处理：用INT\_MAX, INT\_MIN clamp
5. 当字符串里出先其它非法字符，怎样处理：允许数字串之后有其它字符，直接忽略
6. 空格和‘\t'是否忽略，还是当作非法字符：允许string开始有空格和'\'
7. 如果字符串不以合法子串开头怎么处理：返回0

**Sol: modularization**

分3步实现

1. 扫描开始的空格，指针移到第一个不是空格的位置。
2. 检查下一个位置是否为加减号，如果是，保存当前的符号并移动到下一个位置。
3. 循环扫描一个数字，并累加到结果。但在累加前检查是否溢出，如果可能溢出，则根据正负号返回INT\_MAX或INT\_MIN。

复杂度为O(n)

|  |
| --- |
| **int myAtoi(string str)** {  int k = str.find\_first\_not\_of(" \t\n"); // skip white spaces    bool pos = true; //set sign  if (str[k] == '+' || str[k] == '-') pos = (str[k++] == '+');  //scan numerics until hitting max/min, or non-numerical characters  //TODO: check if str[k] is a digit, if it isn't input is invalid  //in the case we return 0 with invalid input, the handling can be ignored  long long val=0;  for (; k<str.size() && str[k]>='0' && str[k]<='9'; ++k) {  val = val\*10 + str[k]-'0';  if (val > INT\_MAX) return pos? INT\_MAX : INT\_MIN;  }    return pos?val:-val;  } |

# 9. Palindrome Number

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

**Sol: math ops**

reverse当前的数直至它的中位，把反转结果rx与剩余数remainder的前半段比较。如果数字有奇数位，则remainder/10 = rx时为回文数。如果有偶数位则两者相等时为回文数。

**Corner case:** 尾数为0的数除了0外都不是回文数，负数都不是回文数。

复杂度O(n)

|  |
| --- |
| **bool isPalindrome(int x)** {  if (x < 0 || (x > 0 && x % 10 == 0)) return 0;  int rx = 0;  for (;x > rx; x/=10) rx = rx\*10+x%10;  return (x==rx || x==rx/10);  } |

# 10. Regular Expression Matching

|  |
| --- |
| Implement regular expression matching with support for '.' and '\*'. '.' Matches any single character. '\*' Matches zero or more of the preceding element. The matching should cover the entire input string (not partial). |
| Some 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 |

**Corner cases:**  
(1) 字符串以\*开始

(2) 字符串中有连续的\*

(3) 问清楚".\*"表示的意思 (这里假设.\*表示任意连续个'.'，见倒数第二个例子）

**Sol1: c++ standard template library**(cheat)

|  |
| --- |
| #include <regex>  **bool isMatch(string s, string p)** {  return regex\_match(s, regex(p));  } |

**Sol2:recursion**

如果p[1]!='\*'，则s的首字母和p[0]匹配，否则尝试各种将s的前缀匹配p[0]\*的可能。这个算法是指数复杂度的，因为当p 形如'a\*b\*c\*'时，会检查各种可能的对s的划分。

|  |
| --- |
| **bool isMatch(string s, string p)** {  return isMatch\_recur(s, p, 0, 0);  }  inline bool match\_char(char cs, char cp) { return (cs==cp) || (cp == '.'); }  bool isMatch\_recur(string s, string p, int si, int pi) {  if (pi == p.size()) return si == s.size();  if (pi + 1 < s.size() && p[pi+1] == '\*') {  do {  if (isMatch\_recur(s, p, si, pi+2)) return true;  } while (si < s.size() && match\_char(s[si++], p[pi]));  return false;  }  else return match\_char(s[si], p[pi]) && isMatch\_recur(s, p, si+1, pi+1);  } |

**Sol3: DP**

用一个矩阵matched纪录s的长度为m的前缀和p的长度为n的前缀匹配的可能性。这样的好处是遇到'\*‘时，如果要更新matched[m][n]，只要检查matched[m][n-2]和matched[m][n-1].

这样该算法的时间复杂度是O(MN), M = s.size(), N = p.size()。

|  |
| --- |
| **bool isMatch(string s, string p)** {  vector<vector<bool>> matched(s.size()+1, vector<bool>(p.size()+1, false));  matched[0][0] = true;  if (p.size()>0 && p[0] == '\*') return false; //invalid pattern  for (int n = 1; n <= p.size(); ++n) {  if (p[n-1] == '\*') {  for (int m = 0; m <= s.size(); ++m)  matched[m][n] = matched[m][n-2] ||  (m > 0 && matched[m-1][n] && match\_char(s[m-1], p[n-2]));  }  else {  for (int m = 1; m <= s.size(); ++m)  matched[m][n] = matched[m-1][n-1] && match\_char(s[m-1], p[n-1]);  }  }  return matched[s.size()][p.size()];  }  inline bool match\_char(char cs, char cp) { return cs==cp || cp == '.'; } |

以上算法空间复杂度是O(MN)。但其实我们只需要最后三行，因此可以优化空间复杂度到O(N)

|  |
| --- |
| **bool isMatch(string s, string p)** {  if (p.size()>0 && p[0] == '\*') return false; //invalid pattern  vector<bool> matched0(s.size()+1, false); matched0[0] = true;  vector<bool> matched1(s.size()+1, false); matched1[1] = match\_char(s[0], p[0]);  vector<bool> matched2(s.size()+1, false);    for (int n = 2; n <= p.size(); ++n) {  if (p[n-1] == '\*') {  for (int m = 0; m <= s.size(); ++m)  matched2[m] = matched0[m] ||  (m > 0 && matched2[m-1] && match\_char(s[m-1], p[n-2]));  }  else {  matched2[0] = false;  for (int m = 1; m <= s.size(); ++m)  matched2[m] = matched1[m-1] && match\_char(s[m-1], p[n-1]);  }  if (n < p.size()) {  swap(matched0, matched1);  swap(matched1, matched2);  }  else return matched2[s.size()];  }  return p.size() == 0? matched0[s.size()] : matched1[s.size()];  }  inline bool match\_char(char cs, char cp) { return cs==cp || cp == '.'; } |

# 

# 

# 11. Container with Most Water

|  |
| --- |
| Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water. |
| Note: You may not slant the container. |

**Sol1: naive enumeration**

考虑任意两个边界i, j，圈定的长方形宽为j-i，高为min(ai, aj)，通过穷举两个边界可以找到最大的容量 area(i, j) = min(ai, aj) \*(j-i)。

|  |
| --- |
| **int maxArea1(vector<int>& height)** {  int max\_area = 0;  for (int j = 0; j < height.size(); ++j)  for (int i = 0; i < j; ++i)  max\_area = max(max\_area, min(a[i], a[j])\*(j-i));  return max\_area;  } |

**Sol2: tail recursion -> iteration**

递归法。假设a0 <= aN。因为对所有n < N, 有area(0, n) < area(0, N)。所以maxArea(0, N) = max(area(0, N), maxArea(1, N))。问题规模每缩小1需要一次运算。所以复杂度为O(N)。

|  |
| --- |
| **int maxArea2(vector<int>& height)** {  int lo = 0, hi = height.size()-1;  int max\_area = (hi - lo) \* min(height[lo], height[hi]);  while (lo < hi) {  if (height[lo] < height[hi]) lo++;  else hi--;  max\_area = max(max\_area, (hi - lo) \* min(height[lo], height[hi]));  }  return max\_area;  } |

# 12. Integer to Roman

|  |
| --- |
| Given an integer, convert it to a roman numeral.Input is guaranteed to be within the range from 1 to 3999. |

**Note:** 罗马数字的规律是

(1) 整五整十有对应

1 5 10 50 100 500 1000

I V X L C D M

(2)四和九形如IV, IX，零是空字符串。

(3)高位在前低位在后。

**Sol1: naive for loop**

每次处理一位。该位对应的字符串是对应10, 5, 1的字符的排列，例如对应10位的是CLX。对9和4进行特殊处理，其余数字拆成5和1的组合。

|  |
| --- |
| **string intToRoman(int num)** {  string syms("MDCLXVI"), result("");  for (int base = 1000, p1 = 0; num > 0; num %= base, base/=10, p1+=2) {  int bit = num/base;  switch (bit) {  case 9: result += syms[p1]; result += syms[p1 - 2]; break;  case 4: result += syms[p1]; result += syms[p1 - 1]; break;  default:  if (bit >= 5) {result += syms[p1 - 1]; bit-=5;}  for (int n = 0; n < bit; ++n) result += syms[p1];  }  }  return result;  } |

**Sol2：greedy**

注意到如果没有4和9，上述实现相当于一个贪婪法的方案。我们可以通过把4和9对应的字符串加入字母表，使得贪婪法对这两种情况也成立。这样可以得到一个简洁的算法。

|  |
| --- |
| **string intToRoman(int num)** {  vector<int> vals = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};  vector<string> syms = vector<string>{"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X",  "IX", "V", "IV", "I"};  string result("");  for (int i = 0; i < vals.size(); ++i)  for (; vals[i] <= num; num -= vals[i]) result += syms[i];  return result;  } |

**Sol3: table lookup**

因为我们只考虑1-3999。因此也可以把每位的10种可能性列在数组里，然后直接拼接。

|  |
| --- |
| **string intToRoman(int num)** {  string M[] = {"", "M", "MM", "MMM"};  string C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};  string X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};  string I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};  return M[num/1000]+C[(num%1000)/100]+X[(num%100)/10]+I[num%10];  } |

# 13. Roman to Integer

|  |
| --- |
| Given a roman numeral, convert it to an integer.  Input is guaranteed to be within the range from 1 to 3999. |

**Sol: math ops**

每次读进一个数，相应地输出加上这个数。唯一的例外是如果当前读进的数大于上一个(例如，IV -> 5-1，则加上current - 2\*last。由于罗马数字的规则不用担心叠加。

|  |
| --- |
| **int romanToInt(string s)** {  int alphabet[256];  alphabet['I'] = 1; alphabet['V'] = 5;  alphabet['X'] = 10; alphabet['L'] = 50;  alphabet['C'] = 100; alphabet['D'] = 500;  alphabet['M'] = 1000;    int val = 0;  for (int n = 0, last = 2000; n < s.size(); ++n) {  int cur = alphabet[s[n]];  val += ((last < cur)? cur - 2\*last : cur);  last = cur;  }  return val;  } |

# 14. Longest Common Prefix

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

**Corner case：**如果array是空的，返回什么？这里假设返回空串。

**Sol: vertical enumeration**

垂直遍历strs[0]的每一个字母，并检查改字母是否出现在所有单词的相应位置。

这个算法的时间复杂度是O(nm)，n是数组长度，m是最短字符串长度。

|  |
| --- |
| **string longestCommonPrefix(vector<string>& strs)** {  if (strs.size() == 0) return "";    for (int i = 0; i < strs[0].size(); ++i) {  for (int n = 1; n < strs.size(); ++n) {  if (i == strs[n].size() || strs[n][i] != strs[0][i]) {  return strs[0].substr(0, i);  }  }  }  return strs[0];  } |

**Note:** Leetcode网站上有其它几个solution。但其实要验证公共前缀就要O(mn)时间，所以没有优化的余地。

# 15. 3Sum

|  |
| --- |
| Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.  Note:  Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)  The solution set must not contain duplicate triplets. |

**Sol: two pointer**

这个题是2sum的延伸，想法是固定最小的数，该数右边的数作2sum。题目的note已经暗示了要先对数组排序。算法复杂度是O(n2)。

因为不允许出现重复，想法是第一、第二个数的搜索要跳过重复的数。

|  |
| --- |
| **vector<vector<int>> threeSum(vector<int>& nums)** {  vector<vector<int>> result;  if (nums.size() < 3) return result;  sort(nums.begin(), nums.end());  for (int i = 0; i < nums.size();) {  int target = -nums[i];  int front = i+1, back = nums.size()-1;  // two sum  while (front < back) {  int sum = nums[front] + nums[back];  if (sum < target) ++front;  else if (sum > target) --back;  else {  result.push\_back(vector<int>{ nums[i], nums[front], nums[back]});  while (nums[++front] == nums[front-1]);  while (nums[--back] == nums[back+1]);  }  }  while (nums[++i] == nums[i-1]); //search next smallest number  }  return result;  } |

# 

# 16. 3Sum Closest

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

**Sol: two pointer**

和上题类似，要化简为2sum来做。区别是需要maintain当前最好的target和距离，且当距离已经为0时及时返回。

|  |
| --- |
| **int threeSumClosest(vector<int>& nums, int target)** {  if (nums.size() < 3) {  throw invalid\_argument("length of input less than 3"); //throw error  }    sort(nums.begin(), nums.end());  int best\_target = nums[0] + nums[1] + nums[2];  int best\_diff = abs(best\_target - target);  for (int n = 0; n < nums.size()-2; ++n) {  int target2 = target - nums[n];  for (int front = n+1, back = nums.size()-1; front < back; ) {  int sum = nums[front] + nums[back];  int diff = abs(sum - target2);  if (abs(sum-target2) < best\_diff) { //update best difference  best\_target = nums[n] + sum;  best\_diff = diff;  }  if (sum == target2) return target; //move front/back pointers  else if (sum < target2) front++;  else back--;  }  }  return best\_target;  } |

# 17. Letter Combinations of a Phone Number

|  |
| --- |
| Given a digit string, return all possible letter combinations that the number could represent.  A mapping of digit to letters (just like on the telephone buttons) is given below.  1() 2(abc) 3(def)  4(ghi) 5(jkl) 6(mno)  7(prqs)8(tuv) 9(wxyz)  0(\_) |

**Analysis:** 给定一个数字串，可以画出一棵多叉树（例如下图表示302对应的树）。第k层表示前k个数（不包括第k个）对应的所有字符串。最后输出的所有字符串是这棵树的叶子结点。这棵树可以通过深度优先或广度优先遍历所有叶子结点。

**Sol1: DFS/recursion**

假定给出数字串前k个数对应的某个串，递归地获得第k个字符是c时对应地所有串，再将所有c的可能性合并起来。

|  |
| --- |
| **vector<string> letterCombinations1(string digits)** {  vector<string> result;  if (digits.size() == 0) return result;  letterCombinations\_recur(digits, string(""), result);  return result;  }  void letterCombinations\_recur(string digits, string prefix, vector<string> &result) {  string alphabet[10] = {" ", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",  "wxyz"};  int n = prefix.length();  if (n == digits.length()) result.push\_back(prefix);  else {  for (int m = 0; m < alphabet[digits[n]-'0'].length(); ++m)  letterCombinations\_recur(digits, prefix+alphabet[digits[n]-'0'][m], result);  }  } |

**Sol2: BFS**

maintain一个保存当前层结点的队列（前k个字符对应的所有串）。向后append一个合法字符得到下一层。迭代到最后一层即得到输出结果。优点是不需要像深度优先一样把顶层的结点压在栈里。

|  |
| --- |
| **vector<string> letterCombinations2(string digits)** {  vector<string> result;  if (digits.size() == 0) return result;    result.push\_back("");    string alphabet[10] = {" ", "", "abc", "def", "ghi", "jkl",  "mno", "pqrs", "tuv", "wxyz"};    for (int i = 0; i < digits.size(); ++i) {  if (digits[i] < '0' || digits[i] > '9')  throw invalid\_argument("invalid input: contains non-digit letters");  string candidate = alphabet[digits[i]-'0'];  vector<string> tmp;  for (int j = 0; j < candidate.size(); ++j) {  for (int k = 0; k < result.size(); ++k) {  tmp.push\_back(result[k] + candidate[j]);  }  }  result.swap(tmp);  }  return result;  } |

# 18. 4Sum

|  |
| --- |
| Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target. |

**Sol：two pointer**

这题是之前的3sum的进一步扩展。每次固定前两个数，对后面的两个数2sum。

|  |
| --- |
| **vector<vector<int>> fourSum(vector<int>& nums, int target)** {  vector<vector<int> > result;  if (nums.size()==0) return result;  sort(nums.begin(), nums.end());  for (int n1 = 0; n1 < nums.size(); n1++) {  for (int n2 = n1 + 1; n2 < nums.size(); n2++) {  int target\_sum = target - nums[n1] - nums[n2];  int front = n2 + 1, back = nums.size() - 1;  while(front < back){  int two\_sum = nums[front] + nums[back];  if (two\_sum < target\_sum) front++;  else if (two\_sum > target\_sum) back--;  else {  vector<int> quadruplet({nums[n1], nums[n2], nums[front], nums[back]});  result.push\_back(quadruplet);  while (front < back && nums[front] == quadruplet[2]) ++front;  while (front < back && nums[back] == quadruplet[3]) --back;  }  }  while(n2 + 1 < nums.size() && nums[n2 + 1] == nums[n2]) ++n2;  }  while (n1 + 1 < nums.size() && nums[n1 + 1] == nums[n1]) ++n1;  }  return result;  } |

# 19. Remove Nth Node from End of List

|  |
| --- |
| Given a linked list, remove the nth node from the end of list and return its head.  For example,  Given linked list: 1->2->3->4->5, and n = 2.  After removing the second node from the end, the linked list becomes 1->2->3->5.  Note:  Given n will always be valid.  Try to do this in one pass. |

**Sol: two pointer**

使用一快一慢两个指针，快指针比慢指针早走k步，这样当快指针指向NULL时慢指针指向要删除的结点。

|  |
| --- |
| **ListNode\* removeNthFromEnd(ListNode\* head, int n)** {  if (head == NULL || n <= 0) return head; //return if empty linked link    // append sentinel node in front of list node to simplify processing  ListNode sentinel(0);  sentinel.next = head;    // move fast pointer to n-th node; if list has less than n nodes, return head  ListNode \*lp\_fast = &sentinel;  for (int i = 0; i < n; ++i) {  if (lp\_fast->next == NULL) return head;  lp\_fast = lp\_fast->next;  }  // move the two pointers simultaneously  ListNode \*lp\_slow = &sentinel;  for (; lp\_fast->next != NULL; lp\_slow = lp\_slow->next, lp\_fast = lp\_fast->next);    lp\_slow->next = lp\_slow->next->next;  return sentinel.next;  } |

# 

# 20. Valid Parentheses

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

**Sol: Stack**

用一个堆栈存放当前还没有被匹配的左括号。每看到一个新的字符，如果它是左括号则进栈，如果是右括号则试图与栈顶的括号匹配，匹配失败则return false。最后如果栈为空则字符串valid。

|  |
| --- |
| **bool isValid(string s)** {  stack<char> stk;  for (char &c : s) {  switch (c) {  case '(': case '[': case '{':  stk.push(c); break;  case ')':  if (stk.empty() || stk.top() != '(') return false;  stk.pop(); break;  case ']':  if (stk.empty() || stk.top() != '[') return false;  stk.pop(); break;  case '}':  if (stk.empty() || stk.top() != '{') return false;  stk.pop(); break;  default:  throw invalid\_argument("input string contains invalid characters");  }  }  return stk.empty();  } |

# 21. Merge Two Sorted Lists

|  |
| --- |
| 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. |

**Sol1: iteration**

比较直观地我们可以maintain输出链表的tail指针，每次插入两个链表中首元素较小的一个，直到其中一个指针为空。此时tail插入两个链表中不为空的那个。初始时我们创建一个sentinel node作为tail node.

|  |
| --- |
| ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {  ListNode sentinel(0);  ListNode \*tail = &sentinel;  while (l1 != NULL && l2 != NULL) {  if (l1->val < l2->val) { tail->next = l1; l1 = l1->next; }  else {tail->next = l2; l2 = l2->next;}  tail = tail->next;  }    if (l1 == NULL) { tail->next = l2;}  else {tail->next = l1;}    l1 = l2 = NULL;  return sentinel.next;  } |

**Sol2: recursion.**

如果两链表之一为空，返回另外一个链表。不然确定当前链表头，它的next指向当前next和另一链表的merge。

**Note:** 这个实现对长链表可能会stack overflow.

|  |
| --- |
| ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {  if (!l1) return l2;  if (!l2) return l1;  if (l1->val < l2->val) { l1 -> next = mergeTwoLists(l1->next,l2); return l1; }  else { l2 -> next = mergeTwoLists(l2->next,l1); return l2; }  } |

# 22. Generate Parentheses

|  |
| --- |
| Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.  For example, given n = 3, a solution set is:  "((()))", "(()())", "(())()", "()(())", "()()()" |

**Sol: DFS**我们要把长度为2n的字符串用(和)填满。这和前面phone numbers的题一样，对应一棵二叉树。(例如下图是n=3的情况)

((()))

(()())

(())()

()()()

()(())

但其中的很多情况是不可能出现合法串的。有希望出现合法串的情况需要满足两个条件：

(1) 在任意位置左括号的个数大于右括号个数。（因此导致有些branch被剪)

(2) 在任意位置左括号个数少于n。因此直接可以指向一个确定性的串（如上图中的长方形结点）

如果作深度优先遍历，可以用递归写，代码会相对简单。

|  |
| --- |
| **vector<string> generateParenthesis(int n)** {  vector<string> result;  dfs(0, 0, n, string(""), result);  return result;  }  void dfs(int nl, int nr, int n, string str, vector<string> &result) {  if (nl == n) {  for (int k = 0; k < nl-nr; ++k) str += ")";  result.push\_back(str);  return;  }  if (nr < nl) dfs(nl, nr+1, n, str+")", result);  if (nl < n) dfs(nl+1, nr, n, str+"(", result);  } |

如果作广度优先遍历，则对每个结点需要保存前缀串和已经出现的左括号个数。

# 

# 

# 23. Merge k Sorted Lists

|  |
| --- |
| 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. |

**Sol1: recursive merging**

两两合并sorted list。直到所有的list都被合并。假设共有n个list node，则复杂度是需要O(nlogk)。

|  |
| --- |
| **ListNode\* mergeKLists(vector<ListNode\*>& lists)** {  if (lists.size() == 0) return NULL;  if (lists.size() == 1) return lists[0];    int n = lists.size()/2;  vector<ListNode\*> lists\_pre(lists.begin(), lists.begin()+n);  vector<ListNode\*> lists\_post(lists.begin()+n, lists.end());    ListNode\* l1 = mergeKLists(lists\_pre);  ListNode\* l2 = mergeKLists(lists\_post);  //merge l1, l2  ListNode sentinel(0);  ListNode \*tail = &sentinel;  while (l1 != NULL && l2 != NULL) {  if (l1->val < l2->val) { tail->next = l1; l1 = l1->next; }  else {tail->next = l2; l2 = l2->next;}  tail = tail->next;  }    if (l1 == NULL) { tail->next = l2;}  else {tail->next = l1;}  return sentinel.next;  } |

**Sol2: heap**

每次从k个list中取head最小的元素加入输出list，并把head指向下一个位置。这可以通过把所有链表放进一个最小堆，堆的comparator由链表头定义。当链表为空时不再插入堆。因为操作的amortized cost是logk因此代价为O(nlogk)

|  |
| --- |
| **ListNode\* mergeKLists(vector<ListNode\*>& lists)** {  vector<ListNode \*> heap;  for (int k = 0; k < lists.size(); ++k) {  if (lists[k]) heap.push\_back(lists[k]);  }  make\_heap(heap.begin(), heap.end(), cmp);  ListNode sentinel(0); ListNode \*tail = &sentinel;  while (!heap.empty()) {  ListNode \*l\_min = heap.front();  pop\_heap(heap.begin(), heap.end(), cmp);  heap.pop\_back();  tail->next = l\_min; tail = l\_min;  l\_min = l\_min->next;  if (l\_min) {  heap.push\_back(l\_min);  push\_heap(heap.begin(), heap.end(), cmp);  }  }  return sentinel.next;  } |

# 24. Swap Nodes in Pairs

|  |
| --- |
| Given a linked list, swap every two adjacent nodes and return its head.  For example,  Given 1->2->3->4, you should return the list as 2->1->4->3.  Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed. |

**Sol：linked list ops**

维护输出链表的tail，每次取输入链表前两个元素，插入输出链表，并更新输出链表tail和输入链表head指针，直到输入链表head为空（已扫到链表尾部）.

如果第2个元素为NULL，则输出链表插入tail，结束。

|  |
| --- |
| **ListNode\* swapPairs(ListNode\* head)** {  ListNode sentinel(0);  ListNode \*tail = &sentinel;  while (head) {  ListNode \*l1 = head, \*l2 = l1->next;  if (l2) {  head = l2->next;  tail->next = l2;  l2->next = l1;  }  else {  head = NULL;  tail->next = l1;  }  tail = l1;  }  tail->next = NULL;  return sentinel.next;  } |

# 25. Reverse Nodes in k group

|  |
| --- |
| Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.  If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.  You may not alter the values in the nodes, only nodes itself may be changed.  Only constant memory is allowed.  For example,  Given this linked list: 1->2->3->4->5  For k = 2, you should return: 2->1->4->3->5  For k = 3, you should return: 3->2->1->4->5 |

**Sol: linkedlist ops**

分为循环内的三步，每个循环处理k个node并插入输出结点：

1. 输入链表的头指针向前走k个位置，以到达下一组链表的头。如果hit链表尾部则把当前的头插入输出链表。
2. 逆转输入链表的前k个node。输出链表的为指针为当前链表的旧头。这是一个循环插入链表头的操作。到新头后结束。
3. (2)的结果插入输出链表尾。最后更新新头为(1)的结果。

|  |
| --- |
| **ListNode\* reverseKGroup(ListNode\* head, int k)** {  ListNode sentinel\_out(0);  sentinel\_out.next = head;  ListNode \*t\_out = &sentinel\_out;  for (ListNode \*next\_head = head; head; head = next\_head) {  // find next head position  for (int n = 0; n < k; n++, next\_head = next\_head->next) {  if (!next\_head) return sentinel\_out.next;  }  // reverse a group of k-nodes, the last node links to next head  ListNode \*cur\_tail = next\_head, \*i\_group = head;  for (int n = 0; n < k; ++n) {  ListNode \*tmp = i\_group;  i\_group = i\_group->next;  tmp->next = cur\_tail;  cur\_tail = tmp;  }    // append revered sub-list to tail and update new tail as current head  t\_out->next = cur\_tail;  t\_out = head;  }  return sentinel\_out.next;  } |

# 26. Reverse Duplicates from Sorted Array

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

**Sol: read-write pointers**

维护一对读写指针。只有当读数与前一个数不重复（或index=0)时才写。因为读数总比写好的数多，不用担心互斥。

|  |
| --- |
| **int removeDuplicates(vector<int>& nums)** {  if (nums.size() == 0) return 0;  int i\_read, i\_write;  for (i\_read = 0, i\_write = 0; i\_read < nums.size()-1; ++i\_read) {  if (nums[i\_read] != nums[i\_read+1]) nums[i\_write++] = nums[i\_read];  }  nums[i\_write++] = nums[i\_read];  return i\_write;  } |

# 27. Remove Element

|  |
| --- |
| 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. |

**Sol: read-write pointers**

也是维护速度不同的读写指针。只是写指针前进的条件取决于当前读到的值。

|  |
| --- |
| **int removeElement(vector<int>& nums, int val)** {  int i\_write = 0;  for (i\_read = 0; i\_read < nums.size(); ++i\_read) {  if (nums[i\_read] != val) nums[i\_write++] = nums[i\_read];  }  return i\_write;  } |

# 

# 28. Implement strStr

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

**Sol1: enumeration**

穷举所有needle在haystack中的起点位置。

|  |
| --- |
| **int strStr(string haystack, string needle)** {  int sz\_h = haystack.size(), sz\_n = needle.size();  if (sz\_n ==0) return 0;  for (int first = 0; first <= sz\_h - sz\_n; ++first) {  bool matched = true;  for (int i = 0, j = first; i < sz\_n; ++i, ++j) {  if (haystack[j] != needle[i]) {  matched = false;  break;  }  }  if (matched) return first;  }  return -1;  } |

**Sol2:KMP**（一般面试不需要）

通过分析needle的前后缀结构可以跳过一些起始位置。思想是对needle的前n个字符构成的子串，记录它的最长的匹配前缀和后缀（不包括子串本身，下图中红色、绿色部分）。这样当检查匹配发生出错时，可以把子串的前缀移到对应的后缀处。

haystack

needle

needle的前缀结构partial match table T可以通过递归分析。令LMP[n]，LMS[n]为needle[0...n]的最长满足LMP[n] = LMS[n] && LMP!=needle[0...n]的前后缀。T[n]是LMP[n]的长度

给定LMP[1...n-1]我们从长到短搜索LMP[n]：

1. 如果needle[T[n-1]+1] = needle[n]则显然LMP[n] = LMP[n-1]+needle[n].
2. 否则我们需要找needle[0...n-1]中次短的matching suffix/prefix。这可以通过看LMP[n]的LMP和LMS（下图中深红深绿色部分）。由此递归到第一个更小规模的问题。
3. 如果LMP[n]回退到首字母，则T[n]=0.

x

x

x

x

x

＝

＝

matched prefix & suffix

x

|  |
| --- |
| **int strStr(string haystack, string needle)** {  if (needle.size() == 0) return 0;  if (needle.size() > haystack.size()) return -1;    vector<int> table(needle.size()+1, 0); //partial match table  table[0] = -1; table[1] = 0;  for (int pos = 2, cnd = 0; pos <= needle.size(); ++pos) {  if (needle[pos-1] == needle[cnd]) {  table[pos] = (cnd++)+1;  }  else if (cnd > 0) { //reprocess current location  cnd = table[cnd];  --pos;  }  else table[pos] = 0;  }  int first = 0, len = 0;  while(first + len < haystack.size()) {  if (needle[len] == haystack[first + len]){ // extend right end of matched string  if (len == needle.length()-1) return first;  len++;  }  else { //skip to next potential match  first = first + len - (table[len] > -1? table[len] : -1);  len = table[len] > -1? table[len] : 0;  }  }  return -1; // didn't find  } |

# 29. Divide Two Integers

|  |
| --- |
| Divide two integers without using multiplication, division and mod operator.  If it is overflow, return MAX\_INT. |

**Sol: bit ops**

分析overflow:

(1) divisor = 0. （需要确定0/0的behavior)  
(2) dividend = MIN\_INT, divisor = -1

首先去掉符号。然后迭代地计算divisor, divisor\*2, divisor\*4, ...直到divisor\*2k超过dividend。

从divisor\*2k开始用贪心法从dividend中扣除，并把因子加给输出.

|  |
| --- |
| **int divide(int dividend, int divisor)** {  if (divisor == 0 || (dividend == INT\_MIN && divisor == -1)) return INT\_MAX;  if (divisor == 1) return dividend;    bool pos = (dividend < 0) ^ (divisor < 0);  long long abs\_dividend = labs(dividend);  long long abs\_divisor = labs(divisor);  if (abs\_divisor > abs\_dividend) return 0;    long long left\_most\_digit = 1;  long long product = abs\_divisor;  do {  left\_most\_digit <<= 1; product <<= 1;  } while (product < abs\_dividend);  product >>= 1; left\_most\_digit >>=1;  int result = left\_most\_digit;  int rem = abs\_dividend - product;  while (rem > abs\_divisor){  while (rem < product) {  left\_most\_digit >>= 1;  product >>=1;  }  result += left\_most\_digit;  rem -= product;  }  return pos? -result:result;  } |

# 30. Substring with Concatenation of All Words

|  |
| --- |
| You are given a string, s, and a list of words, that are all of the same length. Find all starting indices of substring(s) in s that is a concatenation of each word in words exactly once and without any intervening characters. |
| For example, given:  s: "barfoothefoobarman" words: ["foo", "bar"]  You should return the indices: [0,9].(order does not matter). |

**Sol:sliding window**

这个问题的关键在于它假设字典里的单词是定长的，因此可以把问题当作一个单元为单词长度的sliding window来做。每次window向右边延伸一个单词，此时分三种情况：

1. 单词不在字典中：那么它不能出现在任何窗口中。清空整个窗口，从单词后面重新开始。
2. 单词在字典中，但已经在窗口中全部出现了：此时为了包括最近的一次出现，第一次出现的位置需要被排除在窗口之外。
3. 单词出现在字典中，且次数少于字典中的次数：此时不用更新起始位置。

上述第(1)(2)种情况更新起始位置时需要相应地更新字典中的词频quota(+1)。

第(2)(3)种情况需要把当前单词quota减1，并检查是否需要输出到结果。

|  |
| --- |
| **vector<int> findSubstring(string s, vector<string>& words)** {  int nwords = words.size(), wordlen = words[0].size();  vector<int> result;    //build dictionary  unordered\_map<string, int> occ;  for (int i = 0; i < nwords; ++i) occ[words[i]]++;  //search concatenation  for (int i = 0; i < wordlen; ++i) { //assume window front = i + wordlen\*k  int first = i, last = i;  while (last + wordlen <= s.size() && first + wordlen\*nwords <=s.size()) {  string word = s.substr(last, wordlen);  auto iword = occ.find(word);  if (iword == occ.end()) { // word not in dictionary: move first after the word  for (; first < last; first+=wordlen) occ[s.substr(first, wordlen)]++;  first = last+wordlen;  }  else {  if (iword->second == 0) { //word used up in dictionary  while (true){  string first\_word = s.substr(first, wordlen);  occ[first\_word]++;  first += wordlen;  if (word.compare(first\_word) == 0) break;  }  }  iword->second--;  if (last - first == (nwords-1) \* wordlen) result.push\_back(first);  }  last += wordlen;  }  for (; first < last; first+=wordlen) occ[s.substr(first, wordlen)]++;  }  return result;  } |

# 31. Next Permutation

|  |
| --- |
| Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.  If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).  The replacement must be in-place, do not allocate extra memory.  Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.  1,2,3 → 1,3,2  3,2,1 → 1,2,3  1,1,5 → 1,5,1 |

**Sol: recursion->iteration**

注意到如果某位右边的子序列有next permutation，则只需要permutate那个子序列。当右边子序列没有next permutation（即全部逆序排列），则需要得到该位（假设值为n）＋右边序列的next permutation. 想法是找到右边子序列中第一个大于n的数m，替代该位，并将剩下的数升序排列。具体操作时可以先找到m，将m和n交换（此时m右边逆序），然后将右侧子序列的逆转。

|  |
| --- |
| **void nextPermutation(vector<int>& nums)** {  if (nums.size() <=1) return;  //find right most number without a descending seq. to its right  int n, m;  for (n = nums.size()-2; n >= 0 && nums[n] >= nums[n+1]; n--);  //swap m and n  if (n >= 0) {  for (m=n+1; nums[m]>nums[n] && m < nums.size(); ++m);  swap(nums[n], nums[m-1]);  }  // reverse n+1 ... end  for (int l = n+1, r = nums.size()-1; l<r; ++l, --r) swap(nums[l], nums[r]);  } |

# 32. Longest valid parenthesis

|  |
| --- |
| Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.  For "(()", the longest valid parentheses substring is "()", which has length = 2.  Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4. |

**Sol: Stack**

考虑当验证一个括号组时（见20题Valid Parentheses），使用一个栈来检查当前输入是否可能合法。这里的不同之处在于在一个合法子串之前栈内已经有一些东西了，以及只有左括号被push进栈。每次退栈时，如果栈空，则从开始到当前位置为一个合法串，否则，从栈顶元素以后到当前位置为合法串。所以这里我们压入左括号的位置，当出现右括号但栈为空时，我们把“栈底”元素对应的位置更新为当前右括号位置。

|  |
| --- |
| **int longestValidParentheses(string s)** {  stack<int> left\_pos;//stack of unmatched left parenthesis positions  int bottom = -1, maxlen = 0;  for (int n = 0; n < s.size(); ++n) {  if (s[n] == '(') {  left\_pos.push(n);  }  else {  if (left\_pos.empty()) bottom = n;  else {  left\_pos.pop();  maxlen = max(maxlen, n - (left\_pos.empty()? bottom: left\_pos.top()));  }  }  }  return maxlen;  } |

# 33. Search in Rotated Sorted Array

|  |
| --- |
| Suppose a sorted array is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).  You are given a target value to search. If found in the array return its index, otherwise return -1. You may assume no duplicate exists in the array. |

**Sol: binary search**

考虑rotated sorted array中的一段[am, …, an]，一种可能是它代表[am, an]区间，另一种可能是它代表(-inf, am] U [an, inf)区间。因此t落在[am, …, an]的条件是：

1. am < t < an
2. an < am < t || t < an < am

这样我们可以扩展二分法的实现到rotate sorted array

|  |
| --- |
| inline bool in\_range(int left, int right, int target) {  return (left <= target && target <=right) || //for regular sorted array  (right < left && (target <= right || target >= left)); //for rotated sorted array  }  int search(vector<int>& nums, int target) {  if (nums.size() == 0) return -1;  int left = 0, right = nums.size()-1;  while (left < right) {  int mid = (left+right)/2;  if (nums[mid] == target) return mid;  if (in\_range(nums[left], nums[mid], target)) right = mid-1;  else left = mid+1;  }  return nums[left] == target? left:-1;  } |

# 34. Search for a Range

|  |
| --- |
| Given a sorted array of integers, find the starting and ending position of a given target value.  Your algorithm's runtime complexity must be in the order of O(log n).  If the target is not found in the array, return [-1, -1]. |
| Given [5, 7, 7, 8, 8, 10] and target value 8,  return [3, 4]. |

**Sol: binary search**

：首先搜索最左边的target，然后搜索最右边的target.

最左边的target满足nums[n] == target && (n==0 || nums[n-1] < target)

最右边的target满足nums[n] == target && (n==nums.size()-1 || nums[n+1] > target)

二分搜索时如果mid不满足条件，但nums[mid] == target，第一轮向左搜索，第二轮向右搜索

|  |
| --- |
| // init\_left: <0 if searching for the left most target, otherwise equals the left most target position  **inline int search(const vector<int>& nums, int target, int init\_left)** {  int left = max(init\_left, 0), right = nums.size()-1;  while (left < right) {  int mid = (left + right)/2;  if (nums[mid] > target) right = mid-1;  else if (nums[mid] < target) left = mid+1;  else {  if (init\_left < 0) {  if (mid == 0 || nums[mid-1] < target) return mid;  right = mid-1;  }  else {  if (mid == nums.size() || nums[mid+1] > target) return mid;  left = mid+1;  }  }  }  return nums[left] == target? left : -1;  }  vector<int> searchRange(vector<int>& nums, int target) {  if (nums.size() == 0) return vector<int>{-1, -1};  int first = search(nums, target, -1);  int last = (first == -1)? -1: search(nums, target, first);  return vector<int>{first, last};  } |

# 35. Search Insert Position

|  |
| --- |
| Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.  You may assume no duplicates in the array. |

**Sol: binary search**

这其实就是一个标准的二分查找。初始时可能的取值从0...N（N为数组大小）。

折半时，如果target < nums[mid]，可能的取值也包括mid。但如果target > nums[mid]，则可能的取值从mid+1开始。

|  |
| --- |
| **int searchInsert(vector<int>& nums, int target)** {  if (nums.size() == 0) return 0;  int left = 0, right = nums.size();  while (left < right){  int mid = (left+right)/2;  if (target == nums[mid]) return mid;  if (target < nums[mid]) right = mid;  else left = mid+1;  };  return left;  } |

# 36. Valid Sudoku

|  |
| --- |
| Determine if a Sudoku is valid.  The Sudoku board could be partially filled, where empty cells are filled with the character '.'. |

**Sol: trivial for-loop**

给每行、每列和每个block assign一个初始为false的bit table。每观察到一个放了数字的格子检查相应的三个bits，如果它们都为false，则都置true，否则sudoku不valid。

|  |
| --- |
| **bool isValidSudoku1a(vector<vector<char>>& board)** {  const int BLK\_WID = 3; //assume board is 9x9  const int BLK\_SZ = BLK\_WID \* BLK\_WID;  vector< vector<bool> > rows(BLK\_SZ, vector<bool>(BLK\_SZ, false));  vector< vector<bool> > cols(BLK\_SZ, vector<bool>(BLK\_SZ, false));  vector< vector<bool> > blks(BLK\_SZ, vector<bool>(BLK\_SZ, false));    for (int r = 0; r < BLK\_SZ; ++r) {  for (int c = 0; c < BLK\_SZ; ++c) {  if (board[r][c] == '.') continue;  int val = board[r][c] - '1';  int b = (r/BLK\_WID) \* BLK\_WID + (c/BLK\_WID);  if (rows[r][val] || cols[c][val] || cols[c][val]) return false;  else {  rows[r][val] = true;  cols[c][val] = true;  blks[b][val] = true;  }  }  }  return true;  } |

也可以把每行、每列、每块的结果存在一个整数里，用位运算实现。

|  |
| --- |
| **bool isValidSudoku1b(vector<vector<char>>& board)** {  const int BLK\_WID = 3;  const int BLK\_SZ = BLK\_WID \* BLK\_WID;  vector<int> rows(BLK\_SZ, 0), cols(BLK\_SZ, 0), blks(BLK\_SZ, 0);    for (int r = 0; r < BLK\_SZ; ++r) {  for (int c = 0; c < BLK\_SZ; ++c) {  if (board[r][c] == '.') continue;  int val\_mask = 1 << (board[r][c] - '1');  int b = (r/BLK\_WID) \* BLK\_WID + c/BLK\_WID;  if ((rows[r] & val\_mask) || (cols[c] & val\_mask) || (blks[b] & val\_mask)) {  return false;  }  else {  rows[r] |= val\_mask;  cols[c] |= val\_mask;  blks[b] |= val\_mask;  }  }  }  return true;  } |

# 37. Sudoku Solver

|  |
| --- |
| Write a program to solve a Sudoku puzzle by filling the empty cells.  Empty cells are indicated by the character '.'.  You may assume that there will be only one unique solution. |

**Sol: recursion**

递归地进行以下两步操作：

1. Validate当前填入的数字。如果过程中失败，返回false。
2. 如果发现格子全部填满，返回true.
3. 找到第一个未填的格子，尝试0－9的各种取值，填充剩余的格子由下一层递归解决。

为了提高效率，我们只在第一次作validation，并且maintain validation的结果和未填格子的队列。

当开始填一个新的格子的时候，我们取出这个格子对应的行、列、block的bit mask，取它们的或，并取其中第一个false。这样算法改为：

1. 初始化：各行、各列、各块的bit mask，需要填充的格子
2. 如果当前需要填充的格子为空集，返回true
3. 取第一个格子，通过相应的bit mask取它下一个可能的取值。如果所有取值都已取完还没找到解，返回false。

|  |
| --- |
| **void solveSudoku(vector<vector<char>>& board)** {  const int BLK\_WID = 3, BLK\_SZ = BLK\_WID \* BLK\_WID;    vector<int> rows(BLK\_SZ, 0), cols(BLK\_SZ, 0), blks(BLK\_SZ, 0);  vector<pair<int, int>> cells;    init\_mask(board, rows, cols, blks, cells, BLK\_WID, BLK\_SZ);  if (!solveSudoku\_recur(board, rows, cols, blks, cells, BLK\_WID, BLK\_SZ))  throw invalid\_argument("no solution found!");  }  void init\_mask(vector<vector<char>> &board,  vector<int> &rows, vector<int> &cols, vector<int> &blks,  vector<pair<int, int>> &cells, int BLK\_WID, int BLK\_SZ) {  for (int r = 0; r < BLK\_SZ; ++r) {  for (int c = 0; c < BLK\_SZ; ++c) {  if (board[r][c] == '.') {  cells.push\_back(pair<int, int>(r, c));  }  else {  int val\_mask = 1 << (board[r][c] - '1');  int b = (r/BLK\_WID) \* BLK\_WID + c/BLK\_WID;  if ((rows[r] & val\_mask) | (cols[c] & val\_mask) | (blks[b] & val\_mask))  throw invalid\_argument("invalid input board!");  rows[r] |= val\_mask; cols[c] |= val\_mask; blks[b] |= val\_mask;  }  }  }  }  bool solveSudoku\_recur(vector<vector<char>> &board,  vector<int> &rows, vector<int> &cols, vector<int> &blks,  vector<pair<int, int>> &cells, int BLK\_WID, int BLK\_SZ) {  if (cells.empty()) return true;    int r = cells.back().first, c = cells.back().second;  int b= (r/BLK\_WID) \* BLK\_WID + (c/BLK\_WID);    int candidates = rows[r] | cols[c] | blks[b];  cells.pop\_back();    for (int k = 0; k < BLK\_SZ; ++k) {  int val\_mask = 1 << k;  if ((candidates & val\_mask) == 0) {  // set bits and fill in a cell  rows[r] |= val\_mask; cols[c] |= val\_mask; blks[b] |= val\_mask;  board[r][c] = '1' + k;  // solution found  if (solveSudoku\_recur(board, rows, cols, blks, cells, BLK\_WID, BLK\_SZ))  return true;  // restore bits  rows[r] &= (~val\_mask); cols[c] &= (~val\_mask); blks[b] &= (~val\_mask);  }  }  //restore cell  cells.push\_back(pair<int, int>(r, c));  return false;  } |

# 38. Count and Say

|  |
| --- |
| The count-and-say sequence is the sequence of integers beginning as follows: 1, 11, 21, 1211, 111221, ...  1 is read off as "one 1" or 11. 11 is read off as "two 1s" or 21. 21 is read off as "one 2, then one 1" or 1211. Given an integer n, generate the nth sequence.  Note: The sequence of integers will be represented as a string. |

**Sol: recursion->iteration**

实现递归地从上一个“数”推导到下一个“数”的过程：即每次扫描输入串一段重复的数字字符并计数，然后把计数和值写到输出串。

|  |
| --- |
| **string countAndSay(int n)**{  string last("1"), current("");  for (int k = 1; k <n; ++k) {  for (int j = 0, count = 0; j < last.size(); ++j) {  ++count;  if (j+1==last.size() || last[j+1] != last[j]) {  current += ('0'+count);  current += last[j];  count = 0;  }  }  swap(last, current);  current = "";  }  return last;  } |

# 39. Combination Sum

|  |
| --- |
| Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.  The same repeated number may be chosen from C unlimited number of times. |
| Note:  - All numbers (including target) will be positive integers.  - Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).  - The solution set must not contain duplicate combinations.  For example, given candidate set 2,3,6,7 and target 7, a solution set is: [7], [2, 2, 3] |

**Sol: recursion**

首先对数组排序，以保证最后输出的序列是非降序的。这样对组合中的每个数v，它可能的个数是0...target/v个。我们在每个递归里每次固定v的出现次数n，这导致target减少v\*n，组合的前缀增加n个v，问题的规模减1。

|  |
| --- |
| **vector<vector<int>> combinationSum(vector<int>& candidates, int target)** {  vector<vector<int>> result;  if (candidates.size() > 0) {  sort(candidates.begin(), candidates.end());  vector<int> prefix;  combinationSum\_recur(candidates, target, 0, result, prefix);  }  return result;  }  void combinationSum\_recur(const vector<int>& candidates, int target, int first,  vector<vector<int>> &result, vector<int> &prefix) {  if (target == 0) result.push\_back(prefix);  if (first == candidates.size() || candidates[first] > target) return;  int max\_num = target/candidates[first];    for (int n = 0; n <= max\_num; ++n) {  combinationSum\_recur(candidates, target - n \* candidates[first],  first+1, result, prefix);  prefix.push\_back(candidates[first]);  }  for (int n = 0; n <= max\_num; ++n) prefix.pop\_back();  } |

# 40. Combination Sum II

|  |
| --- |
| Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.  Each number in C may only be used once in the combination. |

**Sol: recursion**

这题和上题唯一区别是一个数在组合中最大的出现次数由数组中它重复的次数决定。所以我们通过for循环找到这个数，并确定下一次递归的在candidate中对应的开始位置。

|  |
| --- |
| **vector<vector<int>> combinationSum2(vector<int>& candidates, int target)** {  vector<vector<int>> result;  if (candidates.size() > 0) {  sort(candidates.begin(), candidates.end());  vector<int> prefix;  combinationSum2\_recur(candidates, target, 0, prefix, result);  }  return result;  }  void combinationSum2\_recur(vector<int>& candidates, int target, int index,  vector<int> &prefix, vector<vector<int>> &result) {  if (target == 0) result.push\_back(prefix);  if (index == candidates.size() || candidates[index] > target) return;  int val = candidates[index], count = 0;  while(index + count < candidates.size() && candidates[index+count] == val) count++;    for (int k = 0; k <= count; ++k) {  combinationSum2\_recur(candidates, target-val\*k, index + count, prefix, result);  prefix.push\_back(val);  };  for (int k = 0; k <= count; ++k) prefix.pop\_back();  } |

# 41. First Missing Positive

|  |
| --- |
| Given an unsorted integer array, find the first missing positive integer.  Your algorithm should run in O(n) time and uses constant space. |
| For example,  Given [1,2,0] return 3,  and [3,4,-1,1] return 2. |

**Sol: bucket sort**

用nums存储1...n的occurrence。但我们只在乎1...n是否在nums里出现过，而不是它的频率。因此我们设法把1… n之间的数放在0...n-1的位置，这样我们不用担心mark和原来的数之间产生混淆。

如果这些位置中有数缺失，第一个缺失的数就是我们要找的数。如果没有数缺失，那么第一个缺失的数就是n+1。在这个区域之外的数，或者重复的数可以被别的数替换。

在把值为v的数放进位置为nums[v-1]的过程中，如果当前nums[v-1]在1到n之间且不为v，则它也需要被放进合适的位置。因为每个位置至多被读两次写一次，因此复杂度是O(n).

|  |
| --- |
| **int firstMissingPositive(vector<int>& nums)** {  for (int i = 0; i < nums.size(); ++i) {  for (int val = nums[i]; val > 0 && val <= nums.size() && nums[val-1] != val; )  swap(nums[val-1], val);  }  for (int i = 0; i < nums.size(); ++i) {  if (nums[i] != i+1) return i+1;  }  return nums.size()+1;  } |

# 42. Trapping rain water

|  |
| --- |
| Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining. |
| The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image! |

**Sol1: DP**

每个bin上的水平线的位置是它左边最大值和右边最大值中较小的一个。因此我们为每个bin计算这两个值，从而得到每个bin的水量。

|  |
| --- |
| **int trap(vector<int>& height)** {  vector<int> left;  for (int k = 0, max\_left = 0; k < height.size(); ++k) {  max\_left = max(max\_left, height[k]);  left.push\_back(max\_left);  }    int amount = 0;  for (int k = height.size()-1, max\_right = 0; k>=0; --k) {  max\_right = max(max\_right, height[k]);  amount += (min(max\_right, left[k]) - height[k]);  }  return amount;  } |

**Sol2: two pointer**

假设数组两边有两道高为lmax, rmax的墙，那么只有当找到一对大于min(lmax, rmax)的墙时，水位才会上升。假设lmax < rmax，则我们从左边scan墙高，直到找到大于lmax的墙。此时水位上升到新的lmax, rmax. （在这个两边向中间夹逼的过程中，水位是不会下降的。）

|  |
| --- |
| **int trap(vector<int>& height)** {  int amount = 0;  for (int l = 0, r = height.size()-1, lmax = 0, rmax = 0; l <= r) {  if (height[l] < height[r]) {  lmax = max(lmax, height[l]);  amount += lmax - height[l++];  }  else {  rmax = max(rmax, height[r]);  amount += rmax- height[r--];  }  }  return amount;  } |

# 43. Multiply strings

|  |
| --- |
| Given two numbers represented as strings, return multiplication of the numbers as a string.  Note:  The numbers can be arbitrarily large and are non-negative.  Converting the input string to integer is NOT allowed.  You should NOT use internal library such as BigInteger. |

**Sol: math ops**

从右数第n位的进位由第一个数的第i位和第二个数的第n-i位以及之前n-1个数的进位得到。乘积最多不会超过num1.size()+nums2.size()位。

|  |
| --- |
| **string multiply(string num1, string num2)** {  int l1 = num1.size(), l2 = num2.size();  string result(l1+l2+2, '0');  for (int n = 0, sum = 0; n < l1+l2+2; ++n, sum/=10) {  for (int i = 0, j = n; i <= n; ++i, --j) {  int n1 = i>=l1? 0 : (num1[l1-1-i]-'0');  int n2 = j>=l2? 0 : (num2[l2-1-j]-'0');  sum += n1\*n2;  }  result[l1+l2+1-n] = (sum%10+'0');  }    int pos = result.find\_first\_not\_of("0");  return (pos == string::npos)? string ("0") : result.substr(pos, l1+l2+2-pos);  } |

# 44. Wildcard matching

|  |
| --- |
| Implement wildcard pattern matching with support for '?' and '\*'. |
| '?' Matches any single character. '\*' Matches any sequence of characters (including the empty sequence). The matching should cover the entire input string (not partial).  Some examples: isMatch("aa","a") → false isMatch("aa","aa") → true isMatch("aaa","aa") → false isMatch("aa", "\*") → true isMatch("aa", "a\*") → true isMatch("ab", "?\*") → true isMatch("aab", "c\*a\*b") → false |

**Sol1: recursion**

对p中第一个\*之前的内容，我们可以直接跟s的内容匹配，如果已经发生匹配失败则返回false。如果p中没有\*则直接返回s是否已扫描完。

p中连续出现的\*可以视为一个。如果\*是p的最后一个字符，则可以把s剩下的内容跟\*匹配。

此外的情况需要递归。

|  |
| --- |
| **bool isMatch(string s, string p)** {  return isMatch\_recur(s, p, 0, 0);  }  bool isMatch\_recur(const string &s, const string &p, int si, int pi) {  // match s and p sequentially until finding a "\*" or reaching end of p  while (pi < p.size() && p[pi] != '\*') {  if (si == s.size() || (p[pi] != '?' && p[pi] != s[si])) return false;  ++pi, ++si;  }  if(pi == p.size()) return si == s.size();    // skip continuous \* in p and treat them as a single \*  while (pi <p.size() && p[pi]=='\*') ++pi;  if(pi==p.size()) return true;    for(int i = si; i<s.size(); ++i) { // then do recursion  if(isMatch\_recur(s, p, i, pi)) return true;  }    return false;  } |

**Sol2: DP**

用表格matched[i][j]表示s[0...i]和p[0...j]是否匹配。为了节省空间，我们把matched的最后一行用一个数组存储。

|  |
| --- |
| **bool isMatch(string s, string p)** {  vector<bool> cur(s.size() + 1, false);  cur[0] = true;  for (int j = 1; j <= p.size(); j++) {  bool pre = cur[0]; // use the value before update  cur[0] = cur[0] && p[j - 1] == '\*';  for (int i = 1; i <= s.size(); i++) {  bool temp = cur[i]; // record the value before update  if (p[j - 1] != '\*')  cur[i] = pre && (s[i - 1] == p[j - 1] || p[j - 1] == '?');  else cur[i] = cur[i - 1] || cur[i];  pre = temp; //matched[i-1][j-1]  }  }  return cur[s.size()];  } |

**Sol3: iteration**

注意到递归只发生在\*出现的地方，且递归内部所实现的是匹配一段被\*分隔的串。

我们假设p=p1\*p2\*...\*pn，p1, pn可以为空。我们首先通过去掉p1和pn。

因此我们实际上做的事情是寻找p2, p3, …，pn－1在s中的匹配。并要求对应的子串不相交且保持一样的先后顺序。我们假定p和s有一个匹配，而p2在s中出现的位置包括l1, l2, …。假设在匹配中p2出现的位置在lm，事实上我们也可以把p2放在l1, l2，...lm-1并得到匹配。因此当已经匹配完一个完整的字符串后，不能再发生回退。

我们可以简单修改上面的递归算法来达到避免不必要的回退的目的：我们纪录每层的递归长度并且如果发现递归回退时返回false.

|  |
| --- |
| **bool isMatch(string s, string p)** {  int l = 0;  return isMatch\_recur(s, p, 0, 0, l);  }  bool isMatch\_recur(const string &s, const string &p, int si, int pi, int &rec\_level) {  int cur\_level = rec\_level;    // match s and p sequentially until finding a "\*" or reaching end of p  while (pi < p.size() && p[pi] != '\*') {  if (si == s.size() || (p[pi] != '?' && p[pi] != s[si])) return false;  ++pi, ++si;  }  if(pi == p.size()) return si == s.size();    // skip continuous \* in p and treat them as a single \*  while (pi <p.size() && p[pi]=='\*') ++pi;  if(pi==p.size()) return true;    // recursion  ++rec\_level;  for(int i = si; i<s.size(); ++i) {  if(isMatch\_recur(s, p, i, pi, rec\_level)) return true;  if(recLevel>curLevel+1) return false;  }    return false;  } |

通过以上思想可见递归并不必要，只需要每次遇到\*时保存上一次遇到\*的位置，以便在子串匹配失败的时候回退。

|  |
| --- |
| **bool isMatch(string s, string p)** {  int i = 0, j = 0, asterick = -1, match;  while (i < s.size()) {  if (j < p.size() && p[j] == '\*') { // found a star  match = i;  asterick = j++;  }  else if (j < p.size() && (s[i] == p[j] || p[j] == '?')) { // a generic match  i++;  j++;  }  else if (asterick >= 0) { // match fails, backtrack to last star and its match  i = ++match;  j = asterick + 1;  }  else return false; // no previous position  }  while (j < p.size() && p[j] == '\*') j++;  return j == p.size();  } |

更快、更直观的做法是直接用修改的KMP寻找下一个子串出现的位置。这样每次的代价是O(m\*#ast)

# 45. Jump Game II

|  |
| --- |
| Given an array of non-negative integers, you are initially positioned at the first index of the array.  Each element in the array represents your maximum jump length at that position.  Your goal is to reach the last index in the minimum number of jumps. |
| Note: You can assume that you can always reach the last index.  For example:  Given array A = [2,3,1,1,4], the minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.) |

**Sol:DP**

中心思想是把位置划分为至少0步，1步，2步...s步可达的。

令D[s]表示第s步可以到达的最远位置。

1. 起点位置：D[0]=0, D[1] = A[0].
2. 对s>0: D[s+1] = maxD[s-1]<n<=D[s] {n+A(n)}，即至少需要s+1步的位置的上一步必须是需要s步。如果D[s+1] = D[s]，则不能再继续向前走。
3. 返回值s满足：D[s-1]<A.size()-1<=D[s]

下面我们用first, last表示D[s-1]+1和D[s]，并通过扫描[first, last]来更新D[s+1].

|  |
| --- |
| **int jump(vector<int>& nums)** {  int first = 0, last = 0, s = 0, maxd = 0;  while (maxd < nums.size()-1 && first <= last){  for (int n = first; n <=last; ++n)  maxd = max(maxd, n + nums[n]);  first = last+1, last = maxd, ++s;  }  return maxd >= nums.size()-1? s : -1;  } |

这个算法的结果maxd保留了最远可达的位置，也可以用来判断是否能够到达某个位置。

# 46. Permutations

|  |
| --- |
| Given a collection of distinct numbers, return all possible permutations. |
| Example:  [1,2,3] have the following permutations: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1]. |

**Sol1: iteration**

使用nextPermutation. 这里把nextPermutation作一个小修改以便在找到最后一个permutation后返回false.

|  |
| --- |
| **vector<vector<int>> permute(vector<int>& nums)** {  sort(nums.begin(), nums.end());  vector<vector<int>> result;  do { result.push\_back(nums); } while (nextPermutation(nums));  return result;  }  bool nextPermutation(vector<int>& nums) {  //find right most number without a descending seq. to its right  int n, m;  for (n = nums.size()-2; n >= 0 && nums[n] >= nums[n+1]; n--);  //swap m and n  if (n >= 0) {  for (m=n+1; nums[m]>nums[n] && m < nums.size(); ++m);  swap(nums[n], nums[m-1]);  }  else return false;  // reverse n+1 ... end  for (int l = n+1, r = nums.size()-1; l<r; ++l, --r) swap(nums[l], nums[r]);  return true;  } |

**Sol2: recursion**

把第n个数放在第k位，并permute第1到n-1个数。具体实现时可以先递归地计算前n-1个数的permutation，然后把每个数的第k位和第n位交换。递归在n=1时结束。

|  |
| --- |
| vector<vector<int>> permute(vector<int>& nums) {  return permute\_recur(nums, nums.size()-1);  }  vector<vector<int>> permute\_recur(const vector<int> &nums, int n) {  if (n<=0) return vector<vector<int>>{nums};  vector<vector<int>> result;  vector<vector<int>> result\_last = permute\_recur(nums, n-1);  result.insert(result.end(), result\_last.begin(), result\_last.end());  for (int k = 0; k < n; ++k) {  for (int j = 0; j < result\_last.size(); ++j) {  vector<int> v = result\_last[j];  swap(v[k], v[n]);  result.push\_back(v);  }  }  return result;  } |

# 47. Permutation II

|  |
| --- |
| Given a collection of numbers that might contain duplicates, return all possible unique permutations. |
| For example,  [1,1,2] have the following unique permutations: [1,1,2], [1,2,1], and [2,1,1]. |

**Sol:iteration**

使用nextPermutation（和上题的实现完全相同）。

|  |
| --- |
| **vector<vector<int>> permuteUnique(vector<int>& nums)** {  sort(nums.begin(), nums.end());  vector<vector<int>> result;  do { result.push\_back(nums); } while (nextPermutation(nums));  return result;  }  bool nextPermutation(vector<int>& nums) {  //find right most number without a descending seq. to its right  int n, m;  for (n = nums.size()-2; n >= 0 && nums[n] >= nums[n+1]; n--);  //swap m and n  if (n >= 0) {  for (m=n+1; nums[m]>nums[n] && m < nums.size(); ++m);  swap(nums[n], nums[m-1]);  }  else return false;  // reverse n+1 ... end  for (int l = n+1, r = nums.size()-1; l<r; ++l, --r) swap(nums[l], nums[r]);  return true;  } |

**Sol: recursion**

为了避免重复，我们需要在swap时避免swap两个值相同的数。

|  |
| --- |
| **vector<vector<int> > permuteUnique(vector<int> &num)** {  sort(num.begin(), num.end());  vector<vector<int> >result;  permute\_recur(num, 0, num.size(), result);  return result;  }  void permute\_recur(vector<int> num, int i, int j, vector<vector<int> > &result) {  if (i == j-1) {  result.push\_back(num);  return;  }  for (int k = i; k < j; k++) {  if (i != k && num[i] == num[k]) continue;  swap(num[i], num[k]);  permute\_recur(num, i+1, j, result);  }  } |

# 48. Rotate Image

|  |
| --- |
| You are given an n x n 2D matrix representing an image. Rotate the image by 90 degrees (clockwise).  Could you do this in-place? |

**Sol: modularization**

旋转的过程中对4个相应的位置发生置换（如图）。对位置(r, c)，按counter-clockwise方向旋转（即输出矩阵位置对应元素在输入矩阵中的位置），相应为：

(r, c) <-(n-1-c, r)<-(n-1-r, n-1-c)<- (c, n-1-r)。

需要遍历的(r, c)如蓝色部分所示，每次行的两端缩减1,第0列的两端为0, n-2.

|  |
| --- |
| **void rotate(vector<vector<int>>& matrix)** {  const int wid = matrix.size()-1;  for (int c = 0, first = 0, last = wid; first < last; ++c, ++first, --last) {  for (int r = first; r < last; ++r) {  int tmp = matrix[r][c];  matrix[r][c] = matrix[wid-c][r];  matrix[wid-c][r] = matrix[wid-r][wid-c];  matrix[wid-r][wid-c] = matrix[c][wid-r];  matrix[c][wid-r] = tmp;  }  }  } |

# 49. Group Anagrams

|  |
| --- |
| Given an array of strings, group anagrams together.  Note:  For the return value, each inner list's elements must follow the lexicographic order.  All inputs will be in lower-case. |
| [  ["ate", "eat","tea"],  ["nat","tan"],  ["bat"]  ] |

**Sol1: sorting**

想法是把单词内部按照字母序排序，从而anagram有相同的code。然后我们sort code+word的数组，从而输出先按照code排序（因此anagram组合在一起），再按照字母排序。

|  |
| --- |
| **vector<vector<string>> groupAnagrams(vector<string>& strs)** {  vector<string> code\_word;  for (string word : strs) {  string code = word;  sort(code.begin(), code.end());  code\_word.push\_back(code+word);  }    sort(code\_word.begin(), code\_word.end());  vector<vector<string>> result;  string cur\_code;  for (int k = 0; k < code\_word.size(); ++k){  string code = code\_word[k].substr(0, code\_word[k].size()/2);  string word = code\_word[k].substr(code\_word[k].size()/2);  if (k==0 | cur\_code.compare(code) != 0) {  result.push\_back(vector<string>{});  cur\_code = code;  }  result[result.size()-1].push\_back(word);  }  return result;  } |

**Sol2: hashing**

我们也可以用一个unordered\_map结构存储code->anagram的组合。

|  |
| --- |
| **vector<vector<string>> groupAnagrams(vector<string>& strs)** {  unordered\_map<string, vector<string>> code\_word;  for (string word : strs) {  string code = word;  sort(code.begin(), code.end());  if (code\_word.find(code) == code\_word.end()) code\_word[code] = vector<string>{word};  else code\_word[code].push\_back(word);  }    vector<vector<string>> result;  for (auto cw : code\_word) {  vector<string> &grp = cw.second;  sort(grp.begin(), grp.end());  result.push\_back(grp);  }  return result;  } |

# 50. Pow(x, n)

|  |
| --- |
| Implement Pow(x, n) |

**Sol: binary**

把n用二进制表示。即n = k0\*1 + k1\*2 +km\*2m ，k为binary。从而xn=[(x2)0]k0 ＊…＊ [(x2)m]km.

特殊情况：n=0: 返回1;x=0，返回0或infty. n<0, 返回Pow(1/x, －n)。

溢出：为了规避n=INT\_MIN时-n溢出的问题，Pow(1/x, -n) = Pow(1/x, -(n+1))/x.

|  |
| --- |
| **double myPow(double x, int n)** {  if (n == 0) return 1;  if (x == 0) return n < 0? numeric\_limits<double>::max() : 0;  if (n < 0) return myPow(1/x, -(n+1))/x;    double multiplier = x, result = 1;  while (n) {  if (n % 2) result \*= multiplier;  n/=2; multiplier \*= multiplier;  }  return result;  } |