# Shortlisted problems

# Index

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 51 |  |  |  |  |  |
| 52 |  |  |  |  |  |
| 53 |  |  |  |  |  |
| 54 |  |  |  |  |  |
| 55 |  |  |  |  |  |
| 56 |  |  |  |  |  |
| 57 |  |  |  |  |  |
| 58 |  |  |  |  |  |
| 59 |  |  |  |  |  |
| 60 |  |  |  |  |  |
| 61 |  |  |  |  |  |
| 62 |  |  |  |  |  |
| 63 |  |  |  |  |  |
| 64 |  |  |  |  |  |
| 65 |  |  |  |  |  |
| 66 |  |  |  |  |  |
| 67 |  |  |  |  |  |
| 68 |  |  |  |  |  |
| 69 |  |  |  |  |  |
| 70 |  |  |  |  |  |
| 71 |  |  |  |  |  |
| 72 |  |  |  |  |  |
| 73 |  |  |  |  |  |
| 74 |  |  |  |  |  |
| 75 |  |  |  |  |  |
| 76 |  |  |  |  |  |
| 77 |  |  |  |  |  |
| 78 |  |  |  |  |  |
| 79 |  |  |  |  |  |
| 80 |  |  |  |  |  |
| 81 |  |  |  |  |  |
| 82 |  |  |  |  |  |
| 83 |  |  |  |  |  |
| 84 |  |  |  |  |  |
| 85 |  |  |  |  |  |
| 86 |  |  |  |  |  |
| 87 |  |  |  |  |  |
| 88 |  |  |  |  |  |
| 89 |  |  |  |  |  |
| 90 |  |  |  |  |  |
| 91 |  |  |  |  |  |
| 92 |  |  |  |  |  |
| 93 |  |  |  |  |  |
| 94 |  |  |  |  |  |
| 95 |  |  |  |  |  |
| 96 |  |  |  |  |  |
| 97 |  |  |  |  |  |
| 98 |  |  |  |  |  |
| 99 |  |  |  |  |  |
| 100 |  |  |  |  |  |

# 

51. N Queens

|  |
| --- |
| The n-quees puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.  Given an integer n, return all distinct solutions to the n-queens puzzle. |
| Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively. |

**Sol: recursion**

首先n queens肯定存在不同行，不同列中。我们用pos[k]表示queen在第k行的位置，pos是0...n-1的一个排列。我们同时用三个bitset监视是否每列、每主对角线，每副对角线已经被占据。递归过程就是试图在行r的每一列放置queen，是否放置取决于三个bitset是否冲突。

|  |
| --- |
| **vector<vector<string>> solveNQueens(int n)** {  long long bit\_col = (1 << n)-1;  long long bit\_diag1 = (1 << (n\*2-1))-1;  long long bit\_diag2 = (1 << (n\*2-1))-1;  vector<int> rows(n, -1);  vector<vector<string>> solution;  solveNQueens\_recur(0, rows, solution, bit\_col, bit\_diag1, bit\_diag2);  return solution;  }  void solveNQueens\_recur(int r, vector<int> &rows, vector<vector<string>> &solution,  long long bit\_col, long long bit\_diag1, long long bit\_diag2) {    int n = rows.size();  if (r == n) {  // print board and add to results  solution.push\_back(vector<string>(n, string(n, '.')));  for (int k = 0; k < n; ++k) solution.back()[k][rows[k]] = 'Q';  }  else {  // try put queen at row r, col c if possible  for (int c = 0; c < n; ++c) {  long long bc = 1 << c, bd1 = 1 << (r-c+n-1), bd2 = 1 << (r+c);  if ((bit\_col & bc) && (bit\_diag1 & bd1) && (bit\_diag2 & bd2)) {  rows[r] = c;  solveNQueens\_recur(r+1, rows, solution,  bit\_col&~bc, bit\_diag1&~bd1, bit\_diag2&~bd2);  }  }  }  } |

# 52. N Queens II

|  |
| --- |
| Now, instead outputting board configurations, return the total number of distinct solutions. |

**Sol: recursion**

由于不需要输出解，因此不再需要维护51题解中的rows数组和solutions数组。

|  |
| --- |
| **int totalNQueens(int n)** {  long long bit\_col = (1 << n)-1, bit\_diag1 = (1 << (n\*2-1))-1, bit\_diag2 = (1 << (n\*2-1))-1;  int nsol = 0;  totalNQueens\_recur(0, n, nsol, bit\_col, bit\_diag1, bit\_diag2);  return nsol;  }  void totalNQueens\_recur(int r, int n, int &nsol,  long long bit\_col, long long bit\_diag1, long long bit\_diag2) {    if (r == n) { nsol++; return;}  for (int c = 0; c < n; ++c) {  long long bc = 1 << c, bd1 = 1 << (r-c+n-1), bd2 = 1 << (r+c);  if ((bit\_col & bc) && (bit\_diag1 & bd1) && (bit\_diag2 & bd2)) {  totalNQueens\_recur(r+1, n, nsol,  bit\_col&~bc, bit\_diag1&~bd1, bit\_diag2&~bd2);  }  }  } |

# 53. Maximum Subarray

|  |
| --- |
| Find the contiguous subarray within an array (containing at least one number) which has the largest sum. |
| For example, given the array [−2,1,−3,4,−1,2,1,−5,4],  the contiguous subarray [4,−1,2,1] has the largest sum = 6. |

**Sol1：naive enumeration**

穷举数组的左右终点。

**Sol2：cumsum + DP**

从m到n的子数组之和为cumsum(n)-cumsum(m-1)。固定n，则达到最大子数组之和需要最小化 cumsum(m-1)，因此我们维护minmcumsum(m-1)的值和当前数组之和。这个算法的复杂度是O(n).

|  |
| --- |
| **int maxSubArray(vector<int>& nums)** {  int cumsum = 0, mcs = 0, maxsum = INT\_MIN; //mcs: min cumsum  for (int n = 0; n < nums.size(); ++n) {  mcs = min(mcs, cumsum);  cumsum += nums[n];  maxsum = max(maxsum, cumsum - mcs);  }  return maxsum;  } |

**Sol3: divide and conquer**

我们把数组分为两半。则最小子数组可能出现在左一半、或右一半（递归计算），或者被pivot穿过。对第三种情况，我们从pivot为右终点向前搜索最大数组，或从pivot为左终点向后搜索最大数组。

该算法复杂度O(nlogn)

|  |
| --- |
| **int maxSubArray(vector<int>& nums)** {  return maxSubArray\_recur(nums, 0, nums.size());  }  int maxSubArray\_recur(vector<int> &nums, int left, int right) {  if (right == left+1) return nums[left];  int mid = (left+right)/2;  int ms1 = max(maxSubArray\_recur(nums, left, mid), maxSubArray\_recur(nums, mid, right));    int ms2l = INT\_MIN, ms2r = INT\_MIN;  for (int k = mid-1, cumsum = 0; k>=left; --k) {  cumsum += nums[k];  ms2l = max(ms2l, cumsum);  }  for (int k = mid, cumsum = 0; k<right; ++k) {  cumsum += nums[k];  ms2r = max(ms2r, cumsum);  }  return max(ms2l+ms2r, ms1);  } |

# 54. Spiral matrix

|  |
| --- |
| Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order. |
| For example,  Given the following matrix:  [  [ 1, 2, 3 ],  [ 4, 5, 6 ],  [ 7, 8, 9 ]  ]  You should return [1,2,3,6,9,8,7,4,5]. |

**Sol: modularity**

将矩阵分成若干层。第k层对应：

1. 第k行: k 到 n-1-k列（不包括终点）
2. 第n-1-k列：k到m-1-k行
3. 第m-1-k行：n-1-k到k列
4. 第k列：m-1-k到k行。

结束状态:

1. k>n-1-k | k > m-1-k: 访问结束
2. k=n-1-k：访问第k列的第k行到m-1-k行
3. k=m-1-k: 访问第k行的第k行到n-1-k行

|  |
| --- |
| **vector<int> spiralOrder(vector<vector<int>>& matrix)** {  if (matrix.size() == 0) return vector<int>{};  int m = matrix.size(), n = matrix[0].size();  vector<int> result;  for (int k = 0; k <= m-1-k && k <= n-1-k; ++k) {  for (int l = k; l < n-1-k; ++l) result.push\_back(matrix[k][l]);  if (k == m-1-k) {  result.push\_back(matrix[k][n-1-k]);  break;  }  for (int l = k; l < m-1-k; ++l) result.push\_back(matrix[l][n-1-k]);  if (k == n-1-k) {  result.push\_back(matrix[m-1-k][n-1-k]);  break;  }  for (int l = n-1-k; l > k; --l) result.push\_back(matrix[m-1-k][l]);  for (int l = m-1-k; l > k; --l) result.push\_back(matrix[l][k]);  }  return result;  } |

# 55. Jump Game

|  |
| --- |
| Given an array of non-negative integers, you are initially positioned at the first index of the array.  Each element in the array represents your maximum jump length at that position.  Determine if you are able to reach the last index. |
| For example:  A = [2,3,1,1,4], return true.  A = [3,2,1,0,4], return false. |

**Sol:DP**

使用Jump Game II的方法维护最远能达到的距离，不需要维护所需达到的步数。

|  |
| --- |
| **bool canJump(vector<int>& nums)** {  int first = 0, last = 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;  }  return maxd >= nums.size()-1;  } |

只维护能到达的最远距离。在第k步，检查是否k可达，并试图用k+nums[k]更新该距离。

|  |
| --- |
| **bool canJump(vector<int>& nums)** {  for (int k = 0, maxd = 0; k < nums.size(); ++k) {  if (k > maxd) return false;  maxd = max(maxd, k+nums[k]);  }  return true;  } |

从最后一个位置向前推，维护可以到达终点的起点的最早位置。

|  |
| --- |
| **bool canJump(vector<int>& nums)** {  int final = nums.size() - 1;  for (int k = final-1; k >= 0; --k) {  if (k + nums[k] >= final) final = k;  }  return final == 0;  } |

# 56. Merge Intervals

|  |
| --- |
| Given a collection of intervals, merge all overlapping intervals.  For example, given [1,3],[2,6],[8,10],[15,18], return [1,6],[8,10],[15,18]. |

**Sol1: sort + scan**

假如interval的起点已排好序，则我们只需要maintain 当前interval的end，当新出现的一个区间起点处于当前的interval中时，则只需要看区间的终点是否需要更新。否则需要创造一个新的区间。

|  |
| --- |
| **vector<Interval> merge(vector<Interval>& intervals)** {  vector<Interval> result;  if (intervals.size() == 0) return result;  sort(intervals.begin(), intervals.end(),  [](Interval a, Interval b) {return a.start < b.start;});    result.push\_back(intervals[0]);  for (int k = 1; k < intervals.size(); ++k) {  if (intervals[k].start <= result.back().end)  result.back().end = max(result.back().end, intervals[k].end);  else result.push\_back(intervals[k]);  }  return result;  } |

**Sol2：divide and conquer**

把interval分为两组，分别merge，再把两个组合好的区间顺序merge起来。这样每层的merge由剩余的区间个数决定。最坏情况也是O(nlogn)。但这个算法空间代价比较大。

|  |
| --- |
| **vector<Interval> merge(vector<Interval>& intervals)** {  if (intervals.size() == 0) return vector<Interval>{};  return merge\_recur(intervals, 0, intervals.size());  }  vector<Interval> merge\_recur(const vector<Interval>& intervals, int first, int last) {  if (first == last-1) return vector<Interval>{intervals[first]};  int mid = (first + last)/2;  vector<Interval> i1 = merge\_recur(intervals, first, mid);  vector<Interval> i2 = merge\_recur(intervals, mid, last);  vector<Interval> result;  for (int k1 = 0, k2 = 0; k1 < i1.size() || k2 < i2.size();) {  if (k1 == i1.size() || (k2< i2.size() && i2[k2].start < i1[k1].start)) {  if (result.size()>0 && i2[k2].start <= result.back().end)  result.back().end = max(result.back().end, i2[k2].end);  else result.push\_back(i2[k2]);  k2++;  }  else {  if (result.size()>0 && i1[k1].start <= result.back().end)  result.back().end = max(result.back().end, i1[k1].end);  else result.push\_back(i1[k1]);  k1++;  }  }  return result;  } |

# 57. Insert Interval

|  |
| --- |
| Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).  You may assume that the intervals were initially sorted according to their start times. |
| Example 1: Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].  Example 2: Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].  This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10]. |

**Sol1:** 我们把这当作一个merge interval的问题。首先把所有开始位置小于等于newInterval的区间直接加入输出。现在我们加入newInterval，之后我们再加入原先区间组中的区间，直到merge结束。最后我们把剩余的区间加入结果。

|  |
| --- |
| **vector<Interval> insert(vector<Interval>& intervals, Interval newInterval)** {  vector<Interval> result;  int k = 0;  for (; k < intervals.size() && intervals[k].start <= newInterval.start; ++k)  result.push\_back(intervals[k]);    if (result.size()==0 || result.back().end < newInterval.start)  result.push\_back(newInterval);  else result.back().end = max(result.back().end, newInterval.end);    while (k < intervals.size() && intervals[k].start <= result.back().end)  result.back().end = max(result.back().end, intervals[k++].end);    result.insert(result.end(), intervals.begin()+k, intervals.end());    return result;  } |

**Sol2：**另一个更清晰的解释是搜索区间中第一个和最后一个与新区间相交的区间。将它们之前和之后的区间直接加入结果。相交的两个区间分别更新被merge区间的开始和结束位置。

|  |
| --- |
| **vector<Interval> insert(vector<Interval>& intervals, Interval newInterval)** {  int l, r;  for (l = 0; l < intervals.size() && intervals[l].end < newInterval.start; ++l);  for (r = intervals.size()-1; r >= 0 && intervals[r].start > newInterval.end; r--);    if (l < intervals.size() && intervals[l].start< newInterval.end)  newInterval.start = min(newInterval.start, intervals[l].start);  if (r >=0 && intervals[r].end > newInterval.start)  newInterval.end = max(newInterval.end, intervals[r].end);    vector<Interval> result;  result.insert(result.end(), intervals.begin(), intervals.begin()+l);  result.push\_back(newInterval);  result.insert(result.end(), intervals.begin()+r+1, intervals.end());  return result;  } |

# 58. Length of Last Word

|  |
| --- |
| Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string. If the last word does not exist, return 0.  Note: A word is defined as a character sequence consists of non-space characters only. |
| For example, given s = "Hello World", return 5. |

**Sol: string op**

从后到前找一个单词即可：首先从后向前找到单词的最后一个字符，然后找到单词前的第一个空格。

|  |
| --- |
| **int lengthOfLastWord(string s)** {  size\_t pos\_last = s.find\_last\_not\_of(' ');  if (pos\_last == string::npos) return 0;  size\_t pos\_first = s.find\_last\_of(' ', pos\_last);  return (pos\_first == string::npos)? pos\_last+1 : pos\_last-pos\_first;  } |

# 59. Spiral Matrix II

|  |
| --- |
| Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order. |
| For example, given n = 3, you should return the following matrix:  [  [ 1, 2, 3 ],  [ 8, 9, 4 ],  [ 7, 6, 5 ]  ] |

**Sol: modularity**

修改Spiral Matrix（第54题）即可：

1. 需要先创建matrix
2. 访问改为写，并通过维护变量c决定写的内容
3. m = n, 导致break只需要保留第一个

|  |
| --- |
| **vector<vector<int>> generateMatrix(int n)** {  vector<vector<int>> matrix = vector<vector<int>>(n, vector<int>(n, 0));  int c = 1;  for (int k = 0; k <= n-1-k; ++k) {  for (int l = k; l < n-1-k; ++l) matrix[k][l] = c++;  if (k == n-1-k) {  matrix[k][n-1-k] = c;  break;  }  for (int l = k; l < n-1-k; ++l) matrix[l][n-1-k] = c++;  for (int l = n-1-k; l > k; --l) matrix[n-1-k][l] = c++;  for (int l = n-1-k; l > k; --l) matrix[l][k] = c++;  }  return matrix;  } |

# 60. Permutation Sequences

|  |
| --- |
| The set [1,2,3,…,n] contains a total of n! unique permutations.  By listing and labeling all of the permutations in order, we get the following sequence (ie, for n = 3):  "123"  "132"  "213"  "231"  "312"  "321"  Given n and k, return the kth permutation sequence.  Note: Given n will be between 1 and 9 inclusive. |

**Sol: simple combinatorial math**

假设k为0-based。我们维护一个字母表，初始时为"123...n"。在从右数第m+1位，确定改位后还有m!种可能性。所以该位的值为字母表的第k/m!个字符。

|  |
| --- |
| **string getPermutation(int n, int k)** {  string alph("123456789");  --k;  int prod = 1;  for (int m = 1; m < n; ++m) prod \*= m;  string result("");  for (int m = n-1; m >0; k %= prod, prod /= m, --m) {  result += alph[k/prod];  alph.erase(alph.begin() + k/prod);  }  return result + alph[0];  } |

# 61. Rotate List

|  |
| --- |
| Given a list, rotate the list to the right by k places, where k is non-negative. |
| For example: given 1->2->3->4->5->NULL and k = 2, return 4->5->1->2->3->NULL. |

**Sol: linked list op**

找到list的倒数第k个node，然后把两个子链表的顺序置换。

len-k

k

head

tail

p

new head

|  |
| --- |
| **ListNode\* rotateRight(ListNode\* head, int k)** {  if (head == NULL) return NULL;  int len = 1;  ListNode \*p = head, \*tail = head;    for (; tail->next != NULL; tail = tail->next, ++len);  k %= len;  if (k == 0) return head;  for (int i = 0; i < len-k-1; ++i, p = p->next);  tail->next = head;  head = p->next;  p->next = NULL;  return head;  } |

# 62. Unique Paths

|  |
| --- |
| A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).  The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).  How many possible unique paths are there? |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| start |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  | finish |

**Sol: simple combinatorial math**

需要从左上角出发向下走m-1步，向右走n-1步。可能性有C(m+n-2, n-1).

|  |
| --- |
| **int uniquePaths(int m, int n)** {  if (m > n) return uniquePaths(n, m);  long long result = 1;  for (int k = m+n-2, l = 1; l < m; --k, ++l) result = result \*k/l;  return result;  } |

# 63. Unique Path II

|  |
| --- |
| Follow up for "Unique Paths":  Now consider if some obstacles are added to the grids. How many unique paths would there be?  An obstacle and empty space is marked as 1 and 0 respectively in the grid. |
| For example, there is one obstacle in the middle of a 3x3 grid as illustrated below.  [  [0,0,0],  [0,1,0],  [0,0,0]  ]  The total number of unique paths is 2.  Note: m and n will be at most 100. |

**Sol: DP**

用npath[i][j]表示从左上角到(i,j)位置的可能性。事实上我们只需要maintain最后一行。

|  |
| --- |
| **int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid)** {  if (obstacleGrid.size() == 0) return 0;  int nrow = obstacleGrid.size(), ncol = obstacleGrid[0].size();    vector<int> npath(ncol, 0);  npath[0] = 1;  for (int r = 0; r < nrow; ++r) {  npath[0] = obstacleGrid[r][0]==1? 0 : npath[0];  for (int c = 1; c < ncol; ++c) {  npath[c] = obstacleGrid[r][c]==1? 0 : npath[c-1]+npath[c];  }  }  return npath.back();  } |

# 64. Minimum Path Sum

|  |
| --- |
| Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.  Note: You can only move either down or right at any point in time. |

**Sol: DP**

类似上题，用minpath[i][j]表示到位置(i, j)的最小path sum.

|  |
| --- |
| **int minPathSum(vector<vector<int>>& grid)** {  if (grid.size()==0) return 0;  int nrow = grid.size(), ncol = grid[0].size();    vector<int> minpath(ncol, INT\_MAX);  minpath[0] = 0;  for (int m = 0; m < nrow; ++m) {  minpath[0] += grid[m][0];  for (int n = 1; n < ncol; ++n) {  minpath[n] = min(minpath[n-1], minpath[n]) + grid[m][n];  }  }  return minpath.back();  } |

# 65. Valid Number

|  |
| --- |
| Validate if a given string is numeric. |
| Some examples:  "0" => true  " 0.1 " => true  "abc" => false  "1 a" => false  "2e10" => true  Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one. |

**Clarifications:**

面试时需要问清下面几个case

1. 字符串首位可以有空格，中间不可以有空格
2. 字符串的格式是(+/-)x.y e(+/-)n
3. x,y不可以同时为空串，n不可以为空串
4. x, n可以以多个0开始，y可以以0结束

**Sol:modularity**

首先扫描掉开始和结束的空串

optional:扫描+/-号

扫描一段数字串，并记录它的长度

optional:扫描"."，再扫描一段数字串，并积累数字串长度

如果数字串总长度为0,返回false

optional:扫描e,然后optional扫描+/-号和一个非空数字串。

如果此时到达尾部则返回true，否则返回false

|  |
| --- |
| **bool isNumber(string s)** {  // strip leading and trailing spaces  size\_t pos = s.find\_first\_not\_of(" ");  size\_t pos\_last = s.find\_last\_not\_of(" ");  if (pos == string::npos) return false; // if all chars are 0  int ndigit1 = 0, ndigit2 = 0, ndigit3 = 0;  // scan first possible signs and following digit string  if (s[pos] == '-' || s[pos] == '+') ++pos;  ndigit1 = scanDigits(s, pos, pos\_last);  if (pos > pos\_last) return ndigit1>0;  // scan fraction part  if (s[pos] == '.') {  ++pos;  ndigit2 = scanDigits(s, pos, pos\_last);  }  if (ndigit1 + ndigit2 == 0) return false;  if (pos > pos\_last) return true;  // scan exponential part  if (s[pos] == 'e') {  ++pos;  if (s[pos] == '-' || s[pos] == '+') ++pos;  ndigit3 = scanDigits(s, pos, pos\_last);  if (pos > pos\_last) return ndigit3 > 0;  }  return false;  }  inline int scanDigits(const string &s, size\_t &pos, size\_t &pos\_last) {  int ndigits = 0;  while (pos <= pos\_last && s[pos] >= '0' && s[pos] <= '9') {  ++pos, ++ ndigits;  }  return ndigits;  } |

# 66. Plus One

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

**Sol math ops**

找到最后一个不是9的数。把这个数＋1,之后的数全部置0。如果数全是9,则插入1,其余数置0.

|  |
| --- |
| **vector<int> plusOne(vector<int>& digits)** {  int k = digits.size()-1;  while (k >= 0 && digits[k] == 9) --k;  if (k < 0) {  vector<int> result(digits.size()+1, 0);  result[0] = 1;  return result;  }  else {  vector<int> result(digits);  ++result[k];  fill(result.begin()+k+1, result.end(), 0);  return result;  }  } |

# 67. Add Binary

|  |
| --- |
| Given two binary strings, return their sum (also a binary string).  For example,  a = "11"  b = "1"  Return "100". |

**Sol: math ops**

和的位数最多为a和b的位数加1。结果的每位为a,b和进位之和。

|  |
| --- |
| **string addBinary(string a, string b)** {  int sz\_a = a.size(), sz\_b = b.size();  int sz = max(sz\_a, sz\_b) + 1;  string result(sz, '0');  for (int i = sz-1, i1 = sz\_a-1, i2 = sz\_b-1, carry = 0; i >= 0; --i, --i1, --i2) {  int va = (i1 >=0)? a[i1]-'0' : 0;  int vb = (i2 >=0)? b[i2]-'0' : 0;  int sum = va + vb + carry;  result[i] = sum%2 + '0';  carry = sum/2;  }  size\_t pos = result.find\_first\_not\_of("0");  return pos == string::npos? "0":result.substr(pos);  } |

# 68. Text Justification

|  |
| --- |
| Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.  You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.  Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.  For the last line of text, it should be left justified and no extra space is inserted between words. |
| For example,  words: ["This", "is", "an", "example", "of", "text", "justification."]  L: 16.  Return the formatted lines as:  [  "This is an",  "example of text",  "justification. "  ]  Note: Each word is guaranteed not to exceed L in length. |

**Sol: string ops + modularization**

首先定位本行的第一个单词first和最后一个单词last，总共字母数为nchar，空格数为L-nchar。前t个空格为s+1格，后last - first - t为s格。

如果last为最后一个单词，则s=1，t＝0.否则s = L-nchar/(last-first), t = L-nchar-(last-first)\*s.

结束的空格数为L-nchar-(last-first)\*s - t.

|  |
| --- |
| **vector<string> fullJustify(vector<string>& words, int maxWidth)** {  vector<string> result;  for (int first = 0; first < words.size();) {  int nchar = 0, last = first;  while (last < words.size() && nchar+words[last].size()+last-first<= maxWidth)  nchar += words[last++].size();  result.push\_back(transform\_line(words, maxWidth, first, last-1, nchar));  first = last;  }  return result;  }  inline string transform\_line(vector<string>& words, int maxWidth,  int first, int last, int nchar){  int s = (first == last || last == words.size()-1)? 1 : (maxWidth-nchar)/(last-first);  int t = (first == last || last == words.size()-1)? 0 : maxWidth-nchar-(last-first)\*s;  int r = maxWidth - nchar - (last-first)\*s - t;    string str = words[first];  for (int k = 1; first+k<= last; ++k)  str+= string(k <= t? s+1 : s, ' ') + words[first+k];  return str + string(r, ' ');  } |

# 69. Sqrt(x)

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

**Sol: binary search**

搜索范围[1, x]

|  |
| --- |
| **int mySqrt(int x)** {  if (x<=0) return 0;  int left = 1, right = x;  while (left < right-1) {  int mid = (left + right)/2;  if (x/mid < mid) right = mid;  else left = mid;  }  return left;  } |

# 70. Climb Stairs

|  |
| --- |
| You are climbing a stair case. It takes n steps to reach to the top.  Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? |

**Sol: mathematical reduction**

令f(n)是到达第n格的方式，它的上一步可能是第n-1和n-2格。由此可以得到递推公式：

f(n) = f(n-1) + f(n-2)，f(0)=1, f(1)=2。因此它是一个Fibonacci数列。

|  |
| --- |
| **int climbStairs(int n)** {  const double phi = (sqrt(5.0) + 1.0)/2.0;  return (pow(phi, n+1) - pow(1.0-phi, n+1))/sqrt(5.0);  } |

# 

# 

# 71. Simplify Path

|  |
| --- |
| Given an absolute path for a file (Unix-style), simplify it.  For example,  path = "/home/", => "/home"  path = "/a/./b/../../c/", => "/c" |

**corner cases:**

1. 连续出现的"/"：视为一个"/"
2. 若当前目录已经为"/"之后又出现".."，即目录"/../":视之为根目录。

**Sol: stack**

用一个堆栈保存当前的绝对路径，每个元素表示一个子目录。输入路径用"/"分割，分隔出的串若为"."或空则还在当前目录，若为".."则去上层目录（出栈），否则去某一个下层目录（进栈）。结束时若栈为空则结果为"/"，否则结果为所有元素用"/"连接。

|  |
| --- |
| **string simplifyPath(string path)** {  stringstream ss(path);  string cur;  stack<string> folders;  while (getline(ss, cur, '/')) {  if (cur.size() == 0 || cur == ".") continue;  if (cur == "..") {  if (!folders.empty()) folders.pop();  }  else folders.push(cur);  }    string out\_path;  while (!folders.empty()) {  out\_path = "/" + folders.top() + out\_path;  folders.pop();  }  return out\_path.size()==0? "/":out\_path;  } |

# 72. Edit Distance

|  |
| --- |
| Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)  You have the following 3 operations permitted on a word:  a) Insert a character  b) Delete a character  c) Replace a character |

**Sol: DP**

用矩阵D[i][j]表示word1[0...i]和word2[0...j]之间的最小编辑距离。

现在考虑word1的最后一字母i和word2的最后一个字母j,有下列可能性：

1. i和j是对应字母：D[i][j] = D[i-1][j-1] + (word1[i]==word2[j])?0:1;
2. i对应字母j-1: D[i][j] = D[i][j-1]+1
3. j对应字母i-1: D[i][j] = D[i-1][j]+1

|  |
| --- |
| **int minDistance(string word1, string word2)** {  int sz1 = word1.size(), sz2 = word2.size();  vector<int> dist;  for (int k = 0; k <= sz2; ++k) dist.push\_back(k);    for (int j = 1; j <= sz1; ++j) {  int pre = dist[0];  dist[0] = j;  for (int k = 1; k <= sz2; ++k) {  int d = min(pre + (word1[j-1]==word2[k-1]?0:1), min(dist[k-1], dist[k])+1);  pre = dist[k];  dist[k] = d;  }  }  return dist.back();  } |

# 73. Set Matrices Zero

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

**Sol: use input array to save space**

我们需要m+n个flag表示每行或每列是否有0.我们可以把这些元素写到第一行、第一列。由于首行和首列对应的flag都写到0,0位置，我们可以用一个临时变量表示第一行。

在更新时我们需要先更新除掉被用来作flag的元素以外的元素。

面试时如果被问到还是需要说一下这个不是好习惯。

|  |
| --- |
| **void setZeroes(vector<vector<int>>& matrix)** {  if (matrix.size() == 0 || matrix[0].size() == 0) return;  int m = matrix.size(), n = matrix[0].size(), tmp = 1;  for (int k = 0; k < m; ++k) {  for (int j = 0; j < n; ++j) {  if (matrix[k][j] != 0) continue;  if (k==0) tmp = 0; else matrix[k][0] = 0;  matrix[0][j] = 0;  }  }  for (int k = m-1; k >= 0; --k) {  for (int j = n-1; j >= 0; --j) {  if ((k==0 && tmp ==0) || (k > 0 && matrix[k][0] == 0) || matrix[0][j] == 0)  matrix[k][j] = 0;  }  }  } |

# 74. Search in 2D matrix

|  |
| --- |
| Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:   * Integers in each row are sorted from left to right. * The first integer of each row is greater than the last integer of the previous row. |
| For example, consider the following matrix:  [  [1, 3, 5, 7],  [10, 11, 16, 20],  [23, 30, 34, 50]  ]  Given target = 3, return true. |

**Sol: binary search**

这个矩阵按行扫描后是个排过序的矩阵，因此可以用二分搜索。

假定矩阵m行n列，则(i, j)对应的index是i\*m + j，index k对应的位置是(k/n, k%n)

|  |
| --- |
| **bool searchMatrix(vector<vector<int>>& matrix, int target)** {  if (matrix.size() == 0 || matrix[0].size() == 0) return false;  const int M = matrix.size(), N = matrix[0].size();    int left = 0, right = M\*N-1;  while (left <= right) {  int mid = (left+right)/2;  if (matrix[mid/N][mid%N] == target) return true;  if (matrix[mid/N][mid%N] < target) left = mid + 1;  else right = mid - 1;  }  return false;  } |

# 75. Sort Colors

|  |
| --- |
| Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.  Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.  Note:  You are not suppose to use the library's sort function for this problem. |

**Sol1:counting sort**

从左到右数一下有多少0， 1， 2。然后在数组里写相应个数的0，1，2.

|  |
| --- |
| **void sortColors(vector<int>& nums)** {  int count[3] = {0, 0, 0};  for (int k = 0; k < nums.size(); ++k) count[nums[k]]++;  int l = 0;  for (int k = 0; k < 3; ++k) {  for (int j = 0; j < count[k]; ++j, ++l) nums[l] = k;  }  } |

**Sol2: partition**

作两次partition，一次把0放在数组左边，第二次把2放在数组右边。

|  |
| --- |
| **void sortColors(vector<int>& nums)** {  for (int k = 0, left = 0; k < nums.size(); ++k) {  if (nums[k] == 0) {  swap(nums[k], nums[left]);  ++left;  }  }  for (int k = nums.size()-1, right = nums.size()-1; k >= 0; --k) {  if (nums[k] == 2) {  swap(nums[k], nums[right]);  --right;  }  }  } |

更简短的1-pass实现：这里我们把两个partition合并到一起，但需要注意由于归并到右边是从左到右扫描的，因此swap到位置k的是一个未检查过的元素，因此需要检查它是否为0.

|  |
| --- |
| for (int k = 0, left = 0, right = nums.size()-1; k < nums.size(); ++k) {  while (nums[k] == 2 && k < right) swap(nums[k], nums[right--]);  while (nums[k] == 0 && k > left ) swap(nums[k], nums[left++]);  } |

# 76. Minimum Window Substring

|  |
| --- |
| Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).  Note:  If there is no such window in S that covers all characters in T, return the empty string "".  If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S. |
| For example,  S = "ADOBECODEBANC"  T = "ABC"  Minimum window is "BANC". |

**Sol: sliding window**

在S中维护一个sliding window的起点和终点。每次先找到一个终点使得sliding window涵盖了T中所有字母，然后试图缩短起点直到sliding window不能涵盖T中所有的字母。在缩短起点的过程中更新最小窗口。

检查sliding window是否涵盖T：

用一个histogram记录T中的元素在sliding window中还未出现的频率。但如果sliding window中出现的字符频率超过T中的频率，则histogram可以为负。用一个counter记录有多少T中的字符已经出现在sliding window中。当histogram由负变0或反之时，counter不改变。counter为0则表示涵盖了所有字符。

|  |
| --- |
| **string minWindow(string s, string t)** {  vector<int> hist(256, 0);  for (char c : t) ++hist[c];    int count = t.size(), min\_first = 0, min\_len = INT\_MAX;  for (int first = 0, last = 0; last < s.size(); ++ last) {  if (--hist[s[last]] >= 0) count--;  if (count > 0) continue; //current sliding window needs expanding to the right  for (; count == 0; ++first) if (++hist[s[first]] > 0) count++; //shrink leftside  if (last - first + 2 < min\_len) {  min\_len = last - first + 2;  min\_first = first - 1;  }  }  return min\_len == INT\_MAX? "" : s.substr(min\_first, min\_len);  } |

# 77. Combinations

|  |
| --- |
| Given two integers n and k, return all possible combinations of k numbers out of 1 ... n. |
| For example, if n = 4 and k = 2, a solution is:  [  [2,4],  [3,4],  [2,3],  [1,2],  [1,3],  [1,4],  ] |

**Sol: recursion**

递归结构：

求从 m...n的所有combination of k numbers，可以化简为两个case：

1. Combination包括m: 加所有combination在m+1...n之间的combination of k-1
2. Combination不包括：加所有combination在m+1...n之间的combination of k

递归结束条件：

1. k=0 返回已经包含在当前集合里的数。
2. n-m+1<k，没有combinaiton。

|  |
| --- |
| **vector<vector<int>> combine(int n, int k)** {  vector<int> prefix;  vector<vector<int>> result;  combine\_recur(1, n, k, prefix, result);  return result;  }  void combine\_recur(int m, int n, int k, vector<int> &prefix, vector<vector<int>> &result) {  if (n-m+1 < k) return;  if (k == 0) result.push\_back(prefix);  else {  prefix.push\_back(m);  combine\_recur(m+1, n, k-1, prefix, result);  prefix.pop\_back();  combine\_recur(m+1, n, k, prefix, result);  }  } |

# 78. Subsets

|  |
| --- |
| Given a set of distinct integers, nums, return all possible subsets.  Note:  Elements in a subset must be in non-descending order.  The solution set must not contain duplicate subsets. |
| For example, if nums = [1,2,3], a solution is:  [ [3], [1], [2], [1,2,3], [1,3], [2,3], [1,2], [] ] |

**Sol1: recursion**

递归结构：

考虑集合nums[k…]的子集，分两种情况（k包含或不包含在子集中）。如果包含在子集中则把当前prefix放入解中。

递归结束条件：

k == nums.size().

|  |
| --- |
| **vector<vector<int>> subsets(vector<int>& nums)** {  sort(nums.begin(), nums.end());  vector<vector<int>> result;  vector<int> prefix;  result.push\_back(vector<int>{});  subsets\_recur(nums, 0, prefix, result);  return result;  }  void subsets\_recur(const vector<int> &nums, int first, vector<int> &prefix, vector<vector<int>> &result) {  if (first == nums.size()) return;  subsets\_recur(nums, first+1, prefix, result);  prefix.push\_back(nums[first]);  result.push\_back(prefix);  subsets\_recur(nums, first+1, prefix, result);  prefix.pop\_back();  } |

**Sol2: iteration(DP?)**

令Sk为nums[1… k]对应的子集，则令Sk+1为Sk和{(v, nums[k]) | v in Sk}的并。

|  |
| --- |
| **vector<vector<int>> subsets(vector<int>& nums)** {  sort(nums.begin(), nums.end());  vector<vector<int>> result;  result.push\_back(vector<int>{});  for (int k = 0; k < nums.size(); ++k) {  vector<vector<int>> cur = result;  for (vector<int> &v : cur) v.push\_back(nums[k]);  result.insert(result.end(), cur.begin(), cur.end());  }  return result;  } |

**Sol3: bit ops**

如果nums.size()足够小，我们可以把每个子集map到一个unsigned long long, 它的第k位表示nums[k]是否在集合中。

|  |
| --- |
| vector<vector<int>> subsets(vector<int>& nums) {  sort(nums.begin(), nums.end());  vector<vector<int>> result;    unsigned long long nsub = 1 << nums.size();  for (int k = 0; k < nsub; ++k) {  vector<int> cur\_set;  for (int j = 0; j < nums.size(); ++j)  if ((k >> j)%2) cur\_set.push\_back(nums[j]);  result.push\_back(cur\_set);  }  return result;  } |

# 79. Word Search

|  |
| --- |
| Given a 2D board and a word, find if the word exists in the grid.  The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once. |
| For example, given board =  [  ['A','B','C','E'],  ['S','F','C','S'],  ['A','D','E','E']  ]  word = "ABCCED", -> returns true,  word = "SEE", -> returns true,  word = "ABCB", -> returns false. |

**Sol: DFS**

从board的每一个位置开始DFS搜索word。

|  |
| --- |
| **bool exist(vector<vector<char>>& board, string word)** {  if (board.size() == 0 || board[0].size() == 0) return word.empty();  vector<vector<char>> visited(board.size(), vector<char>(board[0].size()));    for (int c = 0; c < board.size(); ++c) {  for (int r = 0; r < board[0].size(); ++r) {  if (exist\_recur(board, word, visited, 0, c, r)) return true;  }  }  return false;  }  bool exist\_recur(const vector<vector<char>> &board, const string word,  vector<vector<char>>& visited, int k, int c, int r) {  if (k == word.size()) return true;  if ( c < 0 || r < 0 || c>= board.size() || r >= board[0].size() ||  visited[c][r]== 1 || board[c][r] != word[k]) {  return false;  }  visited[c][r] = 1;  bool retval = exist\_recur(board, word, visited, k+1, c-1, r) ||  exist\_recur(board, word, visited, k+1, c, r-1) ||  exist\_recur(board, word, visited, k+1, c+1, r) ||  exist\_recur(board, word, visited, k+1, c, r+1);  visited[c][r] = 0;  return retval;  } |

# 80. Remove Duplicates from Sorted Array II

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

**Sol: read-write pointer**

维护一个读指针和一个写指针，并维护一个计数器记录当前数的出现次数。计数器在出现新数时更新为0。写指针会慢于读指针，因此读指针可以保证读到的是输入数组。

|  |
| --- |
| **int removeDuplicates(vector<int>& nums)** {  int iw = 0;  for (int ir = 0, count = 0; ir < nums.size(); ++ir, ++count) {  if (ir > 0 && nums[ir] != nums[ir-1]) count = 0;  if (count < 2) nums[iw++] = nums[ir];  }  return iw;  } |

# 81. Search in Rotated Sorted Array II

|  |
| --- |
| Follow up for "Search in Rotated Sorted Array":  What if duplicates are allowed?  Would this affect the run-time complexity? How and why?  Write a function to determine if a given target is in the array. |

(该题的问题描述不清楚。看API是要求返回boolean表示target是否出现在矩阵中）

**Sol: binary search**

类似之前Search in Rotated Sorted Array一题。

维护搜索范围的左右两边left, right,并另mid = (left+right)/2

（1） A[mid] == target: 已找到

（2） A[mid] < target:

1. target <= A[right] : target必定在右半段
2. target > A[right]: 不确定，但肯定不是right

(3) A[mid] > target:

1. target >= A[left]: target必定在左半段
2. target < A[left]: 不确定，但肯定不是left

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

# 82. Remove Duplicates from Sorted List II

|  |
| --- |
| Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. |
| For example,  Given 1->2->3->3->4->4->5, return 1->2->5.  Given 1->1->1->2->3, return 2->3. |

**Sol: linked list ops**

用一个临时变量last记录上一个被访问的值。如果当前head不为last且head->next==NULL或值与head->val不等时，才放进输出矩阵。

|  |
| --- |
| **ListNode\* deleteDuplicates(ListNode\* head)** {  ListNode sentinel(0);  ListNode \*tail = &sentinel;  for (int last\_val = head==NULL?0:(head->val+1);  head;  last\_val = head->val, head = head->next) {  if (head->val != last\_val && (head->next == NULL || head->val != head->next->val)) {  tail->next = head;  tail = head;  }  }  tail->next = NULL;  return sentinel.next;  } |

# 83. Remove Duplicates from Sorted List

|  |
| --- |
| Given a sorted linked list, delete all duplicates such that each element appear only once. |
| For example,  Given 1->1->2, return 1->2.  Given 1->1->2->3->3, return 1->2->3. |

**Sol: linkedlist ops**

创建输出链表的头指针作为sentinel，遍历输入指针，只有当输入指针与输出指针的tail不同时才能放进链表。因此sentinel的值为head == NULL? 0 : head->val+1 (或任何和head->val不同的值）。

|  |
| --- |
| **ListNode\* deleteDuplicates(ListNode\* head)** {  ListNode sentinel(head == NULL? 0 : (head->val+1));  ListNode \*tail = &sentinel;  for (; head; head = head->next) {  if (head->val != tail->val) {  tail->next = head;  tail = head;  }  }  tail->next = NULL;  return sentinel.next;  } |

# 84. Largest Rectangle in Histogram

|  |
| --- |
| Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram. |
| Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].    The largest rectangle is shown in the shaded area, which has area = 10 unit.  For example, Given heights = [2,1,5,6,2,3], return 10. |

**Sol1: naive enumeration**

假设水位的高度是height[k]，我们需要矩形中所有的shaded bar的height都比它高。因此我们可以向左向右扫描直到出现一个小于height[k]的位置为止。这个算法的复杂度是O(n2).

|  |
| --- |
| **int largestRectangleArea(vector<int>& heights)** {  int max\_area = 0;  for (int k = 0; k < heights.size(); ++k) {  int left = k-1, right = k+1;  while (left > 0 && heights[left] >= heights[k]) --left;  while (right < heights.size() && heights[right] >= heights[k]) ++right;  max\_area = max(max\_area, (right-left-1)\*heights[k]);  }  return max\_area;  } |

**Sol2: stack**

我们现在看有没有可能从左到右扫描histogram的过程中计算出left。这可以通过维护一个stack，并保持stack的内容单调减（即只有小于栈顶元素的元素才能进栈。且访问height[k]时的栈顶元素即为height[k]为bottleneck时对应的left位置）。

|  |
| --- |
| **int largestRectangleArea(vector<int>& heights)** {  stack<int> stk;  vector<int> left(heights.size(), 0);  stk.push(-1);  for (int k = 0; k < heights.size(); ++k) {  while (stk.top() > -1 && heights[k] <= heights[stk.top()]) stk.pop();  left[k] = stk.top();  stk.push(k);  }  while (!stk.empty()) stk.pop();    int max\_area = 0;  stk.push(heights.size());  for (int k = heights.size()-1; k >= 0; --k) {  while (stk.top() < heights.size() && heights[k] <= heights[stk.top()]) stk.pop();  int right = stk.top();  max\_area = max(max\_area, (right - left[k] - 1)\*heights[k]);  stk.push(k);  }  return max\_area;  } |

**Sol3: divide and conquer**

把矩阵拆成左右两部分。这样有三种可能：最大面积完全属于左边，最大面积完全属于右边，最大面积跨越两边。前两种可能性递归到下一层。

第三种可能性，从两边的边界开始，持续加入两侧中较大的那个从而更新面积。这样检查所有可能的水位。

|  |
| --- |
| **int largestRectangleArea(vector<int>& heights)** {  return largestRectangleArea\_recur(heights, 0, heights.size());  }  int largestRectangleArea\_recur(const vector<int>& heights, int left, int right) {  if (right == left) return 0;  if (right == left+1) return heights[left];  int mid = (right + left) / 2;//1    int a1 = largestRectangleArea\_recur(heights, left, mid);  int a2 = largestRectangleArea\_recur(heights, mid, right);    int h = INT\_MAX, a3 = 0;  for (int l = mid-1, r = mid; l>=left || r < right;) {  if (l < left || (r < right && heights[r] > heights[l] )) {  h = min(h, heights[r++]);  }  else {  h = min(h, heights[l--]);  }  a3 = max(a3, (r-l-1)\*h);  }  return max(max(a1, a2), a3);  } |

# 85. Maximal Rectangle

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

**Sol 1:corner enumeration**

穷举矩阵的左上角和右下角，验证是否被1填满，并计算面积。在固定左上角搜索右下角的过程中，在第一次出现0时停止，并检查更新。因此时间是O(m2n2)

**Sol2:row enumeration + stack**

我们考虑第i行为长方形最后一行。并用height记录到从第i行向上连续出现1的个数。然后利用上题我们可以计算第i行为最后一行对应的最大矩阵。因此遍历i行后得到总体的最大矩阵。因为上一题的时间是线性的，因此这个算法的时间是O(mn)

|  |
| --- |
| **int maximalRectangle(vector<vector<char>>& matrix)** {  if (matrix.size() == 0 || matrix[0].size() == 0) return 0;  vector<vector<int>> height(matrix.size(), vector<int>(matrix[0].size(), 0));  int max\_area = 0;  vector<int> heights(matrix[0].size(), 0);  for (int r = 0; r < matrix.size(); ++r) {  for (int c = 0; c < matrix[0].size(); ++c) {  heights[c] = matrix[r][c] == '1'? heights[c]+1 : 0;  }  max\_area = max(largestRectangleArea(heights), max\_area);  }  return max\_area;  }  int largestRectangleArea(vector<int>& heights) {  stack<int> stk;  vector<int> left(heights.size(), 0);  stk.push(-1);  for (int k = 0; k < heights.size(); ++k) {  while (stk.top() > -1 && heights[k] <= heights[stk.top()]) stk.pop();  left[k] = stk.top();  stk.push(k);  }  while (!stk.empty()) stk.pop();  int max\_area = 0;  stk.push(heights.size());  for (int k = heights.size()-1; k >= 0; --k) {  while (stk.top() < heights.size() && heights[k] <= heights[stk.top()]) stk.pop();  int right = stk.top();  max\_area = max(max\_area, (right - left[k] - 1)\*heights[k]);  stk.push(k);  }  return max\_area;  } |

**Sol3:DP**

我们考虑以第i行为底边的长方形，最大长方形的高必为对应第j列height之一。这个长方形的左右边界为第j-height[j]-1行开始每行从第j列开始向左向右搜索第一个出现0的位置。我们用矩阵left, right跟踪这两个位置。

|  |
| --- |
| **int maximalRectangle(vector<vector<char>>& matrix)** {  if (matrix.size() == 0 || matrix[0].size() == 0) return 0;  vector<int> left(matrix[0].size(), 0), right(matrix[0].size(), INT\_MAX),  height(matrix[0].size(), 0);  int max\_area = 0;  for (int i = 0; i < matrix.size(); ++i) {  for (int j = 0, cur\_left = j; j < matrix[0].size(); ++j) {  if (matrix[i][j] == '1') {  height[j]++;  left[j] = max(left[j], cur\_left);  }  else {  height[j] = 0;  left[j] = 0;  cur\_left = j+1;  }  }  for (int j = matrix[0].size()-1, cur\_right = j; j >= 0; --j) {  if (matrix[i][j] == '1') {  right[j] = min(right[j], cur\_right);  max\_area = max(max\_area, (right[j]-left[j]+1)\*height[j]);  }  else {  right[j] = INT\_MAX;  cur\_right = j-1;  }  }  }  return max\_area;  } |

# 86. Partition List

|  |
| --- |
| Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.  You should preserve the original relative order of the nodes in each of the two partitions. |
| For example,  Given 1->4->3->2->5->2 and x = 3,  return 1->2->2->4->3->5. |

**Sol：linkedlist ops**

将输入的内容插入两个链表：一个保存<x的数，另一个保存>=x的数。

|  |
| --- |
| **ListNode\* partition(ListNode\* head, int x)** {  ListNode s1(0), s2(0);  ListNode \*t1 = &s1, \*t2 = &s2;  for (; head; head = head->next) {  if (head->val < x) {  t1->next = head;  t1 = head;  }  else {  t2->next = head;  t2 = head;  }  }  t1->next = s2.next;  t2->next = NULL;  return s1.next;  } |

# 87. Scramble String

|  |
| --- |
| Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.  Below is one possible representation of s1 = "great":  great  / \  gr eat  / \ / \  g r e at  / \  a t  To scramble the string, we may choose any non-leaf node and swap its two children.  For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".  rgeat  / \  rg eat  / \ / \  r g e at  / \  a t  We say that "rgeat" is a scrambled string of "great".  Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".  rgtae  / \  rg tae  / \ / \  r g ta e  / \  t a  We say that "rgtae" is a scrambled string of "great".  Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1. |

**Sol1: recursion + prunning**

我们考虑树的每一层拆分。如果该层发生了拆分，则字符串1的左子树是字符串2右子树的scramble,如果没有发生拆分，则左子树和左子树相对应。为了加速算法，我们在递归的每一层计算字符串的histogram，当histogram不一致时直接返回false。

|  |
| --- |
| **bool isScramble(string s1, string s2)** {  if (s1.size() != s2.size()) return false;  return isScramble\_recur(s1, s2, 0, s1.size(), 0, s2.size());  }  bool isScramble\_recur(const string &s1, const string &s2, int l1, int r1, int l2, int r2) {  if (r1==l1) return true;  if (r1-l1 == 1) return s1[l1] == s2[l2];  vector<int> hist(256, 0);  for (int k = l1; k < r1; ++k) hist[s1[k]]++;  for (int k = l2; k < r2; ++k) hist[s2[k]]--;  for (int h : hist) if (h != 0) return false;    for (int k = l1+1, j = l2+1, l = r2-1; k < r1; ++k, ++j, --l) {  if (isScramble\_recur(s1, s2, l1, k, l2, j)  && isScramble\_recur(s1, s2, k, r1, j, r2)) return true;  if (isScramble\_recur(s1, s2, l1, k, l, r2)  && isScramble\_recur(s1, s2, k, r1, l2, l)) return true;  }  return false;  } |

**Sol2: DP**

用矩阵iss[i][j][l]表示长度为l的子串s1[i...i+l-1], s2[j...j+l-1]是不是互为scramble.

如果l = 1，则取决于s1[i]和s2[j]的比较。

l=k时，考虑k-1种可能的拆分，并通过检索iss比较拆分后子串的匹配。

|  |
| --- |
| **bool isScramble(string s1, string s2)** {  const int L = s1.size();  if (L==0) return true;  if (L==1) return s1[0] == s2[0];    vector<vector<vector<bool>>> iss(L, vector<vector<bool>>(L, vector<bool>(L, false)));  for (int i = 0; i < L; ++i) {  for (int j = 0; j < L; ++j) {  iss[i][j][0] = (s1[i] == s2[j]);  }  }    for (int l = 2; l <= L; ++l) {  for (int i = 0; i+l <= L; ++i) {  for (int j = 0; j+l <= L; ++j) {  for (int k = 1; k < l; ++k) {  if (iss[i][j][k-1] && iss[i+k][j+k][l-k-1]) iss[i][j][l-1] = true;  if (iss[i][j+l-k][k-1] && iss[i+k][j][l-k-1]) iss[i][j][l-1] = true;  }  }  }  }  return iss[0][0][L-1];  } |

# 88. Merge Sorted Array

|  |
| --- |
| Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.  Note:  You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively. |

**Sol: read-write pointer**

这里主要的担心是写元素的时候会覆盖读入的元素。但考虑到一个元素写入的位置不会早于它在输入中的位置，因此我们从右到左作合并。

|  |
| --- |
| void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {  for (int j = m-1, k = n-1, l = m+n-1; j >=0 || k >=0; --l) {  int v1 = j >= 0? nums1[j] : INT\_MIN;  int v2 = k >= 0? nums2[k] : INT\_MIN;  if (v1 > v2) nums1[l] = nums1[j--];  else nums1[l] = nums2[k--];  }  } |

# 89. Gray code

|  |
| --- |
| The gray code is a binary numeral system where two successive values differ in only one bit.  Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0. |
| For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:  00 - 0  01 - 1  11 - 3  10 - 2 |

**Sol: iteration**

最简单的一种Gray code从n= 1: [0, 1]开始，n+1位对应的gray code是在n位的基础上把最高位的0置1，然后从n位gray code的最末元素向前扫描。

|  |
| --- |
| **vector<int> grayCode(int n)** {  vector<int> result;  result.push\_back(0);  if (n==0) return result;  result.push\_back(1);  for (int k = 2, base = 2; k <=n; ++k, base \*= 2) {  for (int j = result.size()-1; j>=0; --j) {  result.push\_back(result[j] + base);  }  }  return result;  } |

# 90. Subsets II

|  |
| --- |
| Given a collection of integers that might contain duplicates, nums, return all possible subsets.  Note:  Elements in a subset must be in non-descending order.  The solution set must not contain duplicate subsets.  For example,  If nums = [1,2,2], a solution is:  [  [2],  [1],  [1,2,2],  [2,2],  [1,2],  []  ] |

**Sol: recursion->iteration**

修改之前的递归或迭代算法：当出现重复数字时，考虑数字出现0….k次的可能性。

下面是迭代算法的实现：

|  |
| --- |
| **vector<vector<int>> subsetsWithDup(vector<int>& nums)** {  sort(nums.begin(), nums.end());  vector<vector<int>> result;  result.push\_back(vector<int>{});  for (int k = 0; k < nums.size();) {  int nr = result.size(), v = nums[k];  for (int base = 0; k < nums.size() && nums[k] == v; ++k, base+=nr) {  for (int j = base; j < base+nr; ++j) {  result.push\_back(result[j]);  result.back().push\_back(v);  }  }  }  return result;  } |

# 91. Decode Ways

|  |
| --- |
| A message containing letters from A-Z is being encoded to numbers using the following mapping:  'A' -> 1  'B' -> 2  ...  'Z' -> 26  Given an encoded message containing digits, determine the total number of ways to decode it.  For example,  Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).  The number of ways decoding "12" is 2. |

**Sol: recursion->iteration**

可以先列出递推式 n[k] = a\*n[k-1] + b\*n[k-2]

a: s[k]是否能转化为1-9

b: s[k-1]=1 或s[k-1]=2 && s[k]为0到6

|  |
| --- |
| **int numDecodings(string s)** {  int pre = 1, last = (s.size() == 0 || s[0] == '0')? 0:1;  for (int k = 1; k < s.size(); ++k) {  int cur = (s[k-1] == '2' && s[k] <= '6' ||s[k-1] == '1' ? pre:0)  + (s[k] > '0' ? last : 0);  pre = last; last = cur;  }  return last;  } |

# 92. Reverse Linked List II

|  |
| --- |
| Reverse a linked list from position m to n. Do it in-place and in one-pass.  Note:  Given m, n satisfy the following condition:  1 ≤ m ≤ n ≤ length of list. |
| For example:  Given 1->2->3->4->5->NULL, m = 2 and n = 4,  return 1->4->3->2->5->NULL. |

**Sol: linked list ops**

维护一个输出list和一个逆转list，两者都需要维护它们的首尾node。把前m-1个node放进输出list。第m-n个node放进逆转list。结束后把这两个list和还未访问的输入list串联起来。

|  |
| --- |
| **ListNode\* reverseBetween(ListNode\* head, int m, int n)** {  ListNode sentinel(0);  ListNode \*tail = &sentinel, \*head\_reverse = NULL, \*tail\_reverse, \*next;  for (int k = 1; k <=n; ++k, head = next) {  next = head->next;  if (k < m) {  tail->next = head;  tail = head;  }  else {  if (head\_reverse == NULL) tail\_reverse = head;  head->next = head\_reverse;  head\_reverse = head;  }  }  tail->next = head\_reverse;  tail\_reverse->next = next;  return sentinel.next;  } |

# 93. Restore IP addresses

|  |
| --- |
| Given a string containing only digits, restore it by returning all possible valid IP address combinations. |
| For example:  Given "25525511135",  return ["255.255.11.135", "255.255.111.35"]. (Order does not matter) |

**Sol: enumeration + prunning**

用一个三层循环即可。每层循环试图转化一个1-3位的数字到0-255.除非数字只有一位不然不能以0开头。并且需要保证第k层循环后还剩(4-k)到12-3k个数字没有转化。

|  |
| --- |
| **vector<string> restoreIpAddresses(string s)** {  vector<string> result;  for (int t1 = 1; t1 <= 3; ++t1) {  if (s.size() - t1 < 3 || s.size() - t1 > 9) continue;  string s1 = substr\_to\_num(s, 0, t1);  if (s1.size() == 0) break;  for (int t2 = t1+1; t2 <= t1+3; ++t2) {  if (s.size() - t2 < 2 || s.size() - t2 > 6) continue;  string s2 = substr\_to\_num(s, t1, t2);  if (s2.size() == 0) break;  for (int t3 = t2+1; t3 <= t2+3; ++t3) {  if (s.size() - t3 < 1 || s.size() - t3 > 3) continue;  string s3 = substr\_to\_num(s, t2, t3);  string s4 = substr\_to\_num(s, t3, s.size());  if (s3.size() > 0 && s4.size() > 0)  result.push\_back(s1+'.'+s2+'.'+s3+'.'+s4);  }  }  }  return result;  }  inline string substr\_to\_num(string &s, int start, int end) {  if (end > start+1 && s[start] == '0') return "";  int val = 0;  for (int k = start; k < end; ++k) val = val\*10 + s[k] - '0';  return (val > 255)? "":s.substr(start, end-start);  } |

# 94. Binary Tree Inorder Traversal

|  |
| --- |
| Given a binary tree, return the inorder traversal of its nodes' values. |
| For example:  Given binary tree {1,#,2,3},  1  \  2  /  3  return [1,3,2]. |

**Sol1:recursion**

按照中序遍历的定义即可。

|  |
| --- |
| **vector<int> inorderTraversal(TreeNode\* root)** {  vector<int> result;  inorderTraversal\_recur(root, result);  return result;  }  void inorderTraversal\_recur(TreeNode \*root, vector<int> &result) {  if (root == NULL) return;  inorderTraversal\_recur(root->left, result);  result.push\_back(root->val);  inorderTraversal\_recur(root->right, result);  } |

**Sol2: iteration**

把root压入栈中,并更新root为左子树。当root为空时访问栈顶元素并把root指向它的右子树。

|  |
| --- |
| **vector<int> inorderTraversal(TreeNode\* root)** {  vector<int> result;  stack<TreeNode \*> stk;  while (!stk.empty() || root) {  if (root) {  stk.push(root);  root = root->left;  }  else {  result.push\_back(stk.top()->val);  root = stk.top()->right;  stk.pop();  }  }  return result;  } |

# 95. Unique Binary Search Trees

|  |
| --- |
| Given n, generate all structurally unique BST's (binary search trees) that store values 1...n. |
| For example,  Given n = 3, your program should return all 5 unique BST's shown below.  1 3 3 2 1  \ / / / \ \  3 2 1 1 3 2  / / \ \  2 1 2 3 |

**Sol: recursion**

递归地构造存储k..j的所有二叉树。遍历所有这棵树所有根结点的可能性。在k<j时，树为NULL.

|  |
| --- |
| **vector<TreeNode\*> generateTrees(int n)** {  if (n==0) return vector<TreeNode\*>{};  return generateTrees\_recur(1, n);  }  vector<TreeNode\*> generateTrees\_recur(int s, int t) {  if (s > t) return vector<TreeNode\*>{NULL};  vector<TreeNode\*> result;    for (int k = s; k <=t; ++k) {  vector<TreeNode\*> v1 = generateTrees\_recur(s, k-1);  vector<TreeNode\*> v2 = generateTrees\_recur(k+1, t);    for (TreeNode \*t1 : v1) {  for (TreeNode \*t2 : v2) {  TreeNode \*r = new TreeNode(k);  r->left = t1;  r->right = t2;  result.push\_back(r);  }  }  }  return result;  } |

# 96. Unique Binary Search Trees

|  |
| --- |
| Given *n*, how many structurally unique BST's (binary search trees) that store values 1...*n*? |
| For example,  Given n = 3, there are a total of 5 unique BST's.  1 3 3 2 1  \ / / / \ \  3 2 1 1 3 2  / / \ \  2 1 2 3 |

**Sol: recursion -> iteration**

用矩阵ntree记录长度为k的数组对应的BST个数。这样不需要在递归过程中建树，而是可以直接迭代。

|  |
| --- |
| **int numTrees(int n)** {  vector<int> ntree(n+1, 0);  ntree[0] = 1;    for (int k = 1; k <=n; ++k)  for (int c = 1; c <=k; ++c)  ntree[k] += ntree[c-1] \* ntree[k-c];    return ntree[n];  } |

# 97. Interleaving string

|  |
| --- |
| Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2. |
| For example,  Given:  s1 = "aabcc",  s2 = "dbbca",  When s3 = "aadbbcbcac", return true.  When s3 = "aadbbbaccc", return false. |

**Solution1: recursion**

用结点(i, j)表示s3[0...i+j]是否是s1[0..i]和s2[0...j]的interleave. 如果(i, j)可以递归到问题(i-1, j)，则相应地在结点之间连一条边。类似地，可以在(i, j)->(i, j-1)之间连接一条边。可见我们寻找解地过程就是在有向图中找(s1.size(), s2.size())->(0,0)路径。

可能的浪费是如果(i, j)已经visit过，说明该问题已经fail，则不需要再搜索。

|  |
| --- |
| **bool isInterleave(string s1, string s2, string s3)** {  if (s1.size() + s2.size() != s3.size()) return false;  vector<vector<bool>> visited(s1.size()+1, vector<bool>(s2.size()+1, false));  return isInterleave\_recur(s1, s2, s3, s1.size(), s2.size(), visited);  }  bool isInterleave\_recur(const string &s1, const string &s2, const string &s3, int n1,  int n2, vector<vector<bool>> &visited) {  if (visited[n1][n2]) return false; else visited[n1][n2] = true;    return (n1 == 0 && n2 == 0)  || (n1 > 0 && s1[n1-1] == s3[n1+n2-1] &&  isInterleave\_recur(s1, s2, s3, n1-1, n2, visited))  || (n2 > 0 && s2[n2-1] == s3[n1+n2-1] &&  isInterleave\_recur(s1, s2, s3, n1, n2-1, visited));  } |

**Solution2: DP**

用矩阵isi[i][j]表示s3[0...i+j]是否是s1[0..i]和s2[0...j]的interleave

|  |
| --- |
| **bool isInterleave(string s1, string s2, string s3)** {  if (s1.size() + s2.size() != s3.size()) return false;    vector<vector<bool>> isi(s1.size()+1, vector<bool>(s2.size()+1, false));  isi[0][0] = true;  for (int k = 0; k < s2.size() && s2[k] == s3[k]; ++k) isi[0][k+1] = true;  for (int k = 0; k < s1.size() && s1[k] == s3[k]; ++k) isi[k+1][0] = true;    for (int k = 1; k <= s1.size(); ++k) {  for (int j = 1; j <= s2.size(); ++j) {  isi[k][j] = (isi[k-1][j] && s1[k-1] == s3[k+j-1])  || (isi[k][j-1] && s2[j-1] == s3[k+j-1]);  }  }  return isi.back().back();  } |

# 98. Validate Binary Search Tree

|  |
| --- |
| Given a binary tree, determine if it is a valid binary search tree (BST).  Assume a BST is defined as follows:  The left subtree of a node contains only nodes with keys less than the node's key.  The right subtree of a node contains only nodes with keys greater than the node's key.  Both the left and right subtrees must also be binary search trees. |

**Sol: in-order traversal**

对树作中序遍历。保证当前访问的数比新访问的数要大。

|  |
| --- |
| **bool isValidBST(TreeNode\* root)** {  TreeNode \*last = NULL;  return inorder\_recur(root, last);  }  bool inorder\_recur(TreeNode \*root, TreeNode \*&last) {  if (root == NULL) return true;  if (!inorder\_recur(root->left, last)) return false;  if (last && (root->val <= last->val)) return false;  last = root;  if (!inorder\_recur(root->right, last)) return false;  return true;  } |

# 99. Recover Binary Search Tree

|  |
| --- |
| Two elements of a binary search tree (BST) are swapped by mistake.  Recover the tree without changing its structure.  Note:  A solution using O(n) space is pretty straight forward. Could you devise a constant space solution? |

**Sol: inorder traversal**

在中序遍历中找到两个逆序数：第一个比后一个数大（该node只更新一次）。第二个比前一个数小（该node一直更新到遍历结束）。交换这两个node的值。

|  |
| --- |
| **void recoverTree(TreeNode\* root)** {  TreeNode \* last = NULL, \*first = NULL, \*second = NULL;  inorder\_recur(root, last, first, second);  assert(first && second);  swap(first->val, second->val);  }  void inorder\_recur(TreeNode \*root, TreeNode \*&last, TreeNode \*&first, TreeNode \*&second) {  if (root == NULL) return;  inorder\_recur(root->left, last, first, second);  if (last && (root->val <= last->val)) {  if (first == NULL) first = last;  second = root;  }  last = root;  inorder\_recur(root->right, last, first, second);  } |

# 100. Same Tree

|  |
| --- |
| Given two binary trees, write a function to check if they are equal or not.  Two binary trees are considered equal if they are structurally identical and the nodes have the same value. |

**Sol: recursion**

按照定义递归即可。

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