# Shortlisted problems

# Index

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 201 | 68 | 56 | Sol1: bit ops (shift)  Sol2: bit trick (xor)  Sol3: bit trick (next power of 2) | easy | interesting appl. for next power of 2 |
| 202 | 4 | 0 | Sol1: fast-slow pointer  Sol2: observe + prove  Sol3: observe + prove | medium | Interesting appl for fast-slow ptr |
| 203 | 20 | 0 | Sol1: linkedlist insert + sentinel  Sol2: linkedlist insert + pointer  Sol3: linkedlist delete + sentinel  Sol4: linkedlist delete + pointer | easy | example of avoiding if condition in linked list, with sentinel or pointer to linked list pointer |
| 204 | 16 | 12 | Sol1: DP  Sol2: DP + code optimization | medium |  |
| 205 | 8 | 8 | Sol: hashing | easy |  |
| 206 | 8 | 8 | Sol: linked list basics | easy |  |
| 207 | 16 | 21 | Sol1: topological sort(BFS)  Sol2: topological sort(DFS) | easy |  |
| 208 | 48 | 60 | Sol: trie core concept | easy |  |
| 209 | 4 | 8 | Sol1: sliding window  Sol2: divide and conquer | easy |  |
| 210 | 24 | 20 | Sol1: topological sort(BFS)  Sol2: topological sort(DFS) | easy |  |
| 211 | 79 | 64 | Sol1: trie(recursion)  Sol2: trie(stack) | medium |  |
| 212 | 68 | 48 | Sol: trie + recursion | medium |  |
| 213 | 0 | 0 | DP |  |  |
| 214 | 8 | 4 | KMP |  |  |
| 215 | 16 | 8 | Quick select |  |  |
| 216 | 0 | 0 | recursion |  |  |
| 217 | 42 | 24 | Sol1: sort  Sol2: unordered\_set |  |  |
| 218 | 49 | 32 | Sol1: line sweeping  Sol2: divide and conquer |  |  |
| 219 | 32 | 12 |  |  |  |
| 220 | 40 | 8 |  |  |  |
| 221 |  |  |  |  |  |
| 222 |  |  |  |  |  |
| 223 |  |  |  |  |  |
| 224 |  |  |  |  |  |
| 225 |  |  |  |  |  |
| 226 |  |  |  |  |  |
| 227 |  |  |  |  |  |
| 228 |  |  |  |  |  |
| 229 |  |  |  |  |  |
| 230 |  |  |  |  |  |
| 231 |  |  |  |  |  |
| 232 |  |  |  |  |  |
| 233 |  |  |  |  |  |
| 234 |  |  |  |  |  |
| 235 |  |  |  |  |  |
| 236 |  |  |  |  |  |
| 237 |  |  |  |  |  |
| 238 |  |  |  |  |  |
| 239 |  |  |  |  |  |
| 240 |  |  |  |  |  |
| 241 |  |  |  |  |  |
| 242 |  |  |  |  |  |
| 243 |  |  |  |  |  |
| 244 |  |  |  |  |  |
| 245 |  |  |  |  |  |
| 246 |  |  |  |  |  |
| 247 |  |  |  |  |  |
| 248 |  |  |  |  |  |
| 249 |  |  |  |  |  |
| 250 |  |  |  |  |  |

# 

# 201. Bitwise AND of Numbers Range

|  |
| --- |
| Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.  For example, given the range [5, 7], you should return 4. |

**Solution1: bit ops (shift)**

考虑每一个bit，只有全部数字改bit都为1,这个bit才为1.

我们比较m和n的每个bit。假如两者从第k位开始不同，则前面k-1位所有数字和m相同。从第k位开始有0有1,因此输出位都为0.

|  |
| --- |
| **int rangeBitwiseAnd(int m, int n)** {  int shift = 0;  while (m != n) {  m >>= 1;  n >>= 1;  ++shift;  }  return m << shift;  } |

**Solution2: bit trick (xor)**

首先取m^n表示两者的不同位diff。我们找到diff的最高位，把m从该位开始置零。

|  |
| --- |
| **int rangeBitwiseAnd(int m, int n)** {  int diff = m ^ n;  int l = 0; //position of highest digit of 1  for (; diff; diff >>= 1) ++l;  return (m >> l) << l;  } |

**Solution3: bit trick (next power of 2)**

一个更有效的办法是利用next power of 2的log(#digit)方法得到从最高位开始全1的bit mask

|  |
| --- |
| **int rangeBitwiseAnd(int m, int n)** {  int diff = m ^ n;    //next power of two minus 1  diff |= diff>>1;  diff |= diff>>2;  diff |= diff>>4;  diff |= diff>>8;  diff |= diff>>16;  return m & ~diff;  } |

**Note:** 此题我能达到的最好running time是68ms。虽然discussion forum上有好几个report 40-50mss的算法，但它们的实际running time在80ms-120ms。从理论上来说最后一个bit trick的worst case复杂度最小 (log (#digits))。

# 

# 202. Happy Number

|  |
| --- |
| Write an algorithm to determine if a number is "happy".  A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.  Example: 19 is a happy number  12 + 92 = 82  82 + 22 = 68  62 + 82 = 100  12 + 02 + 02 = 1 |

**Solution 1: Cycle detection with fast-slow pointer**

我们想象所有的计算结果都放在一个链表里。所以我们是要检测这个链表是否进入了一个环，或会结束。这里我们用之前双指针的solution。

|  |
| --- |
| **bool isHappy(int n)** {  if (n <= 0) return false;  int a = n, b = n;  do {  a = next(a), b = next(next(b));  } while (a != 1 && a != b) ;  return a == 1;  }  inline int next(int n) {  if (n == 1) return 1;  int sum = 0;  for (; n; n/=10) sum += (n%10)\*(n%10);  return sum;  } |

**Solution 2: prove + observe**

考虑验证过程中的总趋势。以a开头的k位数的next最大可能是a\*a+(k-1)\*81,

1. k >= 3: 不可能可能变大。
2. k = 2: 下一个数最多为a\*a+162，当前数至少为a\*100，因此a=1时才可能变大。
3. 能变大的数最多到162. 这时令考虑第2位为b：比较100+b\*10和1+b\*b+81  
   则变大的条件为b = 0 或 1，顺序考虑第3位发现也不可能变大。

这样我们只需要考虑0-100之间的数。用一个bit array来检查改数是否出现过。

|  |
| --- |
| **bool isHappy(int n)** {  if (n < 0) return false;  bool occ[100] = {false};  for (; n > 100 || !occ[n]; n = next(n))  if (n < 100) occ[n] = true;  return n==1;  } |

**Solution 3: prove + observe (more derivation)**

我们可以进一步考虑1－100之间的数ab。 从99开始，持续next（如果>100则再次next ,<100则查表）。直到发现一个小于等于自己的数，这个过程中至多第一次是增加，其它都通过查表得到，因此单调减。如果出现等于当前表头则发现环。如下表，最后我们发现两个环，分别含有1, 4。因此我们只需要持续循环直到发现1或4.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10->1 | 20->4 | 30->9 | 40->16 | 50->25 | 60->36 | 70->49 | 80->64 | 90->81 |
| 1->1 | 11->2 | 21->5 | 31->10 | 41->17 | 51->26 | 61->37 | 71->50 | 81->65 | 91->82 |
| 2->4 | 12->5 | 22->8 | 32->13 | 42->20 | 52->29 | 62->40 | 72->53 | 82->68 | 92->85 |
| 3->9->81->65->61->37  ->20->4 | 13->10 | 23->13 | 33->18 | 43->25 | 53->34 | 63->45 | 73->58 | 83->73 | 93->90 |
| 4->16->37->20->4 | 14->17->4 | 24->20 | 34->25 | 44->32 | 54->41 | 64->52 | 74->65 | 84->80 | 94->97->10 |
| 5->25->20->4 | 15->26->16->4 | 25->29  ->20 | 35->34 | 45->41 | 55->50 | 65->61 | 75->74 | 85->89  ->42 | 95->107->37 |
| 6->36->17->4 | 16->37->20->4 | 26->40->16 | 36->45->41->17 | 46->52->29 | 56->61->37 | 66->72->53 | 76->85->42 | 86->100->1 | 96->117->51 |
| 7->49->10->1 | 17->50->25->20->4 | 27->53->34->25 | 37->58->42->20 | 47->65->61->37 | 57->74->65->61->37 | 67->85->42 | 77->98->42 | 87->113->11 | 97->130->10 |
| 8->64->52->20->4 | 18->65->61->27->20->4 | 28->68->1 | 38->73->58->42->20 | 48->80->64->52->29 | 58->89->42 | 68->100->1 | 78->113->11 | 88->128->69 | 98->145->42 |
| 9->81->65->61->37->4 | 19->82->68->1 | 29->85  ->42  ->20 | 39->90  ->81->65  ->61->37 | 49->97->10 | 59->107->37 | 69->117->51 | 79->130->10 | 89->145->42 | 99->162->41 |

|  |
| --- |
| **bool isHappy(int n)** {  if (n <= 0) return false;  while (n != 1 && n != 4) n = next(n);  return n == 1;  } |

# 203. Remove Linked List Elements

|  |
| --- |
| Remove all elements from a linked list of integers that have value val.  Example  Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6  Return: 1 --> 2 --> 3 --> 4 --> 5 |

**Solution1: linked list insert + sentinel**

input

output

扫描输入链表，把所有不为val的node插入链表尾部。为val的node delete。

需维护的变量：

1. 输入链表的当前链表头head(红色节点）
2. 输出链表的表尾tail（绿色节点）

更新：(head, tail) 两者都非空

如果 head->val !=val => (head->next, tail)

如果 head->val ==val) => (head->next, head)，并设tail[t]->next = head[t]

初始化：head 由输入初始化，tail = &sentinel

|  |
| --- |
| **ListNode\* removeElements(ListNode\* head, int val)** {  ListNode sentinel(0);  for (ListNode\* tail = &sentinel, \* next=NULL; head; head = next) {  next = head->next;  head->next = NULL;  if (head->val != val) {  tail->next = head;  tail = head;  }  else delete head;  }  return sentinel.next;  } |

**Solution2: linked list insert + point referencing**

tail放置output链表尾的next位置。初始化时为head\_out。这样不需要另外创建一个sentinel指针。

input

output

|  |
| --- |
| **ListNode\* removeElements(ListNode\* head, int val)** {  ListNode\* head\_out = NULL;  for (ListNode\*\* tail = &head\_out, \* next=NULL; head; head = next) {  next = head->next;  head->next = NULL;  if (head->val != val) {  \*tail = head;  tail = &(head->next);  }  else delete head;  }  return head\_out;  } |

**Solution3: linked list delete + sentinel**

需维护的变量：上一个链表结点prev和当前结点cur

两种情况：

cur->val!=val: (prev, cur) -> (cur, cur->next)

cur->val==val: (prev, cur) -> (prev, cur->next) 并delete cur

|  |
| --- |
| **ListNode\* removeElements(ListNode\* head, int val)** {  ListNode sentinel(0);  sentinel.next = head;  for (ListNode\* prev = &sentinel, \*cur = head, \* next=NULL; cur; cur = next) {  next = cur->next;  if (cur->val == val) {  prev->next = cur->next;  cur->next = NULL;  delete(cur);  }  else prev = cur;  }  return sentinel.next;  } |

**Solution4: linked list delete + pointer referencing**

|  |
| --- |
| **ListNode\* removeElements(ListNode\* head, int val)** {  ListNode\* head\_out = head, \*next = NULL, \*cur = head;  for (ListNode\*\* prev = &head\_out; cur; cur = next) {  next = cur->next;  if (cur->val == val) {  \*prev = next;  cur->next = NULL;  delete(cur);  }  else prev = &(cur->next);  }  return head\_out;  } |

# 204. Count Primes

|  |
| --- |
| Count the number of prime numbers less than a non-negative number, n. |

**Solution1: DP**

使用排除法。从2开始向n扫描，当发现k是质数，则设所有k\*m(m>=k)的flag为false

|  |
| --- |
| **int countPrimes(int n)** {  vector<bool> primes(n, false);  for (int k = 2; k <sqrt(n); ++k) {  if (!primes[k]) { for (int j = k\*k; j < n; j+=k) primes[j] = true; }  }  return count(primes.begin(), primes.end(), false) - 2;  } |

**Solution2: DP + code optimization**

在上面的基础上我们可以增加一些优化，大大降低计算代价：

* 在DP的同时计算count
* 只考虑所有奇数是否是质数，并从3开始扫描。
* 使用C数组（除非对性能要求非常高，否则最好不要）

|  |
| --- |
| **int countPrimes(int n)** {  if (n <= 2) return 0;  bool\* primes = new bool[n];  int count = n/2;  for (int k = 3; k <sqrt(n); ++k, ++k) {  if (!primes[k]) {  for (int j = k\*k; j < n; j+=k\*2) {  if (!primes[j]) count--;  primes[j] = true;  }  }  }  delete[] primes;  return count;  } |

# 205. Isomophric Strings

|  |
| --- |
| Given two strings s and t, determine if they are isomorphic.  Two strings are isomorphic if the characters in s can be replaced to get t.  All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.  For example,  Given "egg", "add", return true.  Given "foo", "bar", return false.  Given "paper", "title", return true.  Note: You may assume both s and t have the same length. |

**Solution: hashing**

检查是否两个字符串之间存在字母的1－1映射。这需要存储

(1) char->char: 表示s中字符到t中字符的映射

(2) char->bool: 表示t中字符是否已经被映射过

|  |
| --- |
| **bool isIsomorphic(string s, string t)** {  if (s.size() != t.size()) return false;    char charmap[256] = {0};  bool occupied[256] = {false};    for (int k = 0; k < s.size(); ++k) {  if (charmap[s[k]] == 0) {  if (occupied[t[k]] == true) return false;  charmap[s[k]] = t[k];  occupied[t[k]] = true;  }  else {  if (charmap[s[k]] != t[k]) return false;  }  }  return true;  } |

# 206. Reverse Linked List

|  |
| --- |
| Reverse a singly linked list. |

**Solution: linked list basics**

需要维护的指针：

(1)输出的链表头head\_out

(2)输入链表头head

input

output

指针更新：

令head->next = head\_out

(head, head\_out) => (head->next, head)

初始化：head就是输入参数，head\_out=NULL

|  |
| --- |
| **ListNode\* reverseList(ListNode\* head)** {  ListNode\* head\_out = NULL;  while (head) {  ListNode\* next = head->next;  head->next = head\_out;  head\_out = head;  head = next;  }  return head\_out;  } |

# 

# 207. Course Schedule

|  |
| --- |
| There are a total of n courses you have to take, labeled from 0 to n - 1.  Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]  Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?  For example:  2, [[1,0]]  There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.  2, [[1,0],[0,1]]  There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.  Note:  The input prerequisites is a graph represented by a list of edges, not adjacency matrices. |

**Solution: topological sort (BFS)**

构造一个有向图。结点对应课程，边对应先修－后修关系。试图对该图拓扑排序。初始时用unordered\_map<int, list<int>>建立从每一门课到它们后修课程的映射。同时统计所有课的入度。这样我们得到一个入度为0（不要先修课的集合）。遍历所有这些课的后修课，把它们的入度－1, 入度变为0的课程成为后一轮的起始顶点。

我们用一个队列维护入度为0的课程。如果队列为空后还有课程未进队列，则有向图有环，返回false，不然返回true.

|  |
| --- |
| **bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites)** {  // record number of fan-in and fan-out id of each nodes  vector<int> fan\_in(numCourses, 0);  vector<list<int>> adj(numCourses, list<int>{});  for (pair<int, int> p : prerequisites) {  fan\_in[p.second]++;  adj[p.first].push\_back(p.second);  }    // put nodes without fan-in into queue  queue<int> src;  for (int k = 0; k < numCourses; ++k) {  if (fan\_in[k] == 0) src.push(k);  }    // breadth first search  int j = 0;  for (; j < numCourses && !src.empty(); ++j) {  int cur = src.front();  src.pop();  for (int next : adj[cur]) {  if (--fan\_in[next] == 0) src.push(next);  }  }    return j==numCourses;  } |

**Solution: cycle detection(DFS)**

|  |
| --- |
| **bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites)** {  vector<vector<int>> adj = edge\_to\_adjacency(numCourses, prerequisites);  vector<int> visited(numCourses, 0);    for (int k = 0; k < numCourses; ++k) {  if (visited[k] > 0) continue;  if (canFinish\_recur(numCourses, adj, k, visited) == false) return false;  }  return true;  }  bool canFinish\_recur(int numCourses, const vector<vector<int>>& adj, int v,  vector<int>& visited) {  visited[v] = 1;  for (int j : adj[v]) {  if (visited[j] == 1) return false;  if (visited[j] == 0 && canFinish\_recur(numCourses, adj, j, visited) == false)  return false;  }  visited[v] = 2;  return true;  }  vector<vector<int>> edge\_to\_adjacency(int numCourses,  const vector<pair<int, int>>& prerequisites) {  vector<vector<int>> adj(numCourses, vector<int>{});  for (auto& e : prerequisites) adj[e.first].push\_back(e.second);  return adj;  } |

# 208. Implement Trie

|  |
| --- |
| Implement a trie with insert, search, and startsWith methods.  Note: You may assume that all inputs are consist of lowercase letters a-z. |

**Solution: Trie**

Trie是一棵26－叉树。每个结点对应一个字符串前缀，还有一个bool标记该前缀是否是一个单词。

我们实现的函数包括：

1. 构造函数：创建根结点。
2. 析构函数：在TrieNode的析构中递归地删除所有子结点，在Trie中删除根结点即可。
3. Insert: 从根结点开始，每次读一个字符c，并指向下一层，当前结点的第[c-'a']个孩子。如果它为NULL，则创建结点。把最后指向的结点置为is\_word=true.
4. search: 同上述遍历过程。但当遇到NULL时返回false。最后返回最后结点的is\_word
5. startsWidth: 同search，但如遍历完所有字符最后一定返回true.

|  |
| --- |
| **class TrieNode** {  public:  // Initialize your data structure here.  **TrieNode()** {};  **~TrieNode()** { for (int k = 0; k < 26; ++k) delete children[k]; }  TrieNode \*children[26] = {NULL};  bool is\_word = false;  };  class Trie {  public:  **Trie()** { root = new TrieNode(); }    **~Trie()** { delete root; }  // Inserts a word into the trie.  **void insert(string word)** {  TrieNode \*cur = root;  for (char c : word) {  if (cur->children[c-'a'] == NULL) cur->children[c-'a'] = new TrieNode();  cur = cur->children[c-'a'];  }  cur->is\_word = true;  }  // Returns if the word is in the trie.  **bool search(string word)** {  TrieNode \*cur = root;  for (char c : word) {  if (cur->children[c-'a'] == NULL) return false;  cur = cur->children[c-'a'];  }  return cur->is\_word;  }  // Returns if there is any word in the trie  // that starts with the given prefix.  **bool startsWith(string prefix)** {  TrieNode \*cur = root;  for (char c : prefix) {  if (cur->children[c-'a'] == NULL) return false;  cur = cur->children[c-'a'];  }  return true;  }  private:  TrieNode\* root;  };  // Your Trie object will be instantiated and called as such:  // Trie trie;  // trie.insert("somestring");  // trie.search("key"); |

# 209. Minimum Size Subarray Sum

|  |
| --- |
| Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum ≥ s. If there isn't one, return 0 instead.  For example, given the array [2,3,1,2,4,3] and s = 7,  the subarray [4,3] has the minimal length under the problem constraint. |
| If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n). |

**Solution1: sliding window**

维护一个sliding window，和不够时延伸尾段，和太大时缩减头段。这样必然cover所有local最短的和为s的子数组。

|  |
| --- |
| **int minSubArrayLen(int s, vector<int>& nums)** {  int start = 0, last = 0, minlen = INT\_MAX;  for (int sum = 0; last < nums.size(); ++last) {  sum += nums[last];  while (sum >= s) {  minlen = min(minlen, last - start + 1);  sum -= nums[start++];  }  }  return minlen == INT\_MAX? 0:minlen;  } |

**Solution2: prefix sum + binary search**

维护prefix sum数组，每添加一个prefix sum值，检索sum-s+1的lower\_bound,它之前那个数即为window的左边界

|  |
| --- |
| **int minSubArrayLen(int s, vector<int>& nums)** {  vector<int> prefix\_sum(1, 0);  int sum = 0, minlen = INT\_MAX;  for (int v : nums) {  sum += v;  auto low\_it = lower\_bound(prefix\_sum.begin(), prefix\_sum.end(), sum - s+1);  if (low\_it != prefix\_sum.begin()) {  minlen = min(minlen, int(prefix\_sum.end()-low\_it+1));  }  prefix\_sum.push\_back(sum);  }  return minlen == INT\_MAX?0:minlen;  } |

# 210. Course Schedule II

|  |
| --- |
| There are a total of n courses you have to take, labeled from 0 to n - 1.  Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]  Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.  There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.  Note:  The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a graph is represented. |
| For example:  2, [[1,0]]  There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1]  4, [[1,0],[2,0],[3,1],[3,2]]  There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is[0,2,1,3]. |

**Solution: topological sort**

见207题。但返回的是出队列的序列（或如果图有环，返回空数组）。

|  |
| --- |
| **vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)** {    vector<int> order;    // record number of fan-in and fan-out id of each nodes  vector<int> fan\_in(numCourses, 0);  vector<list<int>> adj(numCourses, list<int>{});  for (pair<int, int> p : prerequisites) {  fan\_in[p.first]++;  adj[p.second].push\_back(p.first);  }    // put nodes without fan-in into queue  queue<int> src;  for (int k = 0; k < numCourses; ++k) {  if (fan\_in[k] == 0) src.push(k);  }    // breadth first search  while ( !src.empty() ) {  int cur = src.front();  order.push\_back(cur);  src.pop();  for (int next : adj[cur]) {  if (--fan\_in[next] == 0) src.push(next);  }  }    return order.size()==numCourses? order : vector<int>{};  } |

**Solution: topological sort (DFS)**

|  |
| --- |
| **vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)** {  vector<vector<int>> adj(numCourses, vector<int>{});  for (auto& e : prerequisites) adj[e.first].push\_back(e.second);  vector<int> visited(numCourses, 0);  vector<int> result;    for (int k = 0; k < numCourses; ++k) {  if (visited[k] > 0) continue;  if (findOrder\_recur(numCourses, adj, k, visited, result) == false)  return vector<int>{};  }  return result;  }    bool findOrder\_recur(int numCourses, const vector<vector<int>>& adj, int v,  vector<int>& visited, vector<int>& result) {  visited[v] = 1;  for (int j : adj[v]) {  if (visited[j] == 1) return false;  if (visited[j] == 0 &&  findOrder\_recur(numCourses, adj, j, visited, result) == false)  return false;  }  visited[v] = 2;  result.push\_back(v);  return true;  } |

# 211. Add and Search Word - Data structure

|  |
| --- |
| Design a data structure that supports the following two operations:  void addWord(word)  bool search(word)  search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.  Note: You may assume that all words are consist of lowercase letters a-z. |
| For example:  addWord("bad")  addWord("dad")  addWord("mad")  search("pad") -> false  search("bad") -> true  search(".ad") -> true  search("b..") -> true |

**Solution: trie tree + DFS by stack**

只是在搜索字符为'.'时遍历所有子结点。用DFS实现。

|  |
| --- |
| class TrieNode {  public:  // Initialize your data structure here.  TrieNode() {};  ~TrieNode() {  for (int k = 0; k < 26; ++k) delete children[k];  }  bool is\_word = false;  TrieNode \*children[26] = {NULL};  };  **class WordDictionary {**  **public:**    **WordDictionary()** {  root = new TrieNode();  }    **~WordDictionary()** {  delete root;  }  // Adds a word into the data structure.  **void addWord(string word)** {  TrieNode \*cur = root;  for (char c : word) {  if (cur->children[c-'a'] == NULL) cur->children[c-'a'] = new TrieNode();  cur = cur->children[c-'a'];  }  cur->is\_word = true;  }  // Returns if the word is in the data structure. A word could  // contain the dot character '.' to represent any one letter.  **bool search(string word)** {  stack<pair<TrieNode \*, int>> stk;  stk.push(make\_pair(root, 0));    while (!stk.empty()) {  TrieNode\* node = stk.top().first;  int n = stk.top().second;  stk.pop();  if (n == word.size()) {  if (node->is\_word) return true;  else continue;  }    char c = word[n];  if (c == '.') {  for (char c1 = 'a'; c1 <= 'z'; ++c1) {  if (node->children[c1-'a'] != NULL)  stk.push(make\_pair(node->children[c1-'a'], n+1));  }  }  else {  if (node->children[c-'a'] == NULL) continue;  else stk.push(make\_pair(node->children[c-'a'], n+1));  }  }  return false;  }    private:  TrieNode \*root;  **};** |

**Solution: trie tree + recursion**

和上面的算法一样，但因为使用了系统栈，所以更高效。

|  |
| --- |
| class TrieNode {  public:  // Initialize your data structure here.  TrieNode() {};  ~TrieNode() {  for (int k = 0; k < 26; ++k) delete children[k];  }  bool is\_word = false;  TrieNode\* children[26] = {NULL};  };  **class WordDictionary** {  public:    **WordDictionary()** {  root = new TrieNode();  }    **~WordDictionary()** {  delete root;  }  // Adds a word into the data structure.  **void addWord(string word)** {  TrieNode\* cur = root;  for (char c : word) {  if (cur->children[c-'a'] == NULL)  cur->children[c-'a'] = new TrieNode();  cur = cur->children[c-'a'];  }  cur->is\_word = true;  }  // Returns if the word is in the data structure. A word could  // contain the dot character '.' to represent any one letter.  **bool search(string word)** {  return search\_recur(word, root, 0);  }    **bool search\_recur(const string& word, TrieNode\* node, int first**) {  for (; first < word.size() && word[first] != '.' && node; ++first) {  node = node->children[word[first]-'a'];  }  if (!node) return false;  if (first == word.size()) return node->is\_word;    for (int k = 0; k < 26; ++k) {  if (node->children[k] != NULL && search\_recur(word, node->children[k], first+1))  return true;  }  return false;  }    private:  TrieNode \*root;  }; |

# 212. Word Search II

|  |
| --- |
| Given a 2D board and a list of words from the dictionary, find all words in the board.  Each word must 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 in a word.  Note:  You may assume that all inputs are consist of lowercase letters a-z. |
| For example,  Given words = ["oath","pea","eat","rain"] and board =  [  ['o','a','a','n'],  ['e','t','a','e'],  ['i','h','k','r'],  ['i','f','l','v']  ]  Return ["eat","oath"]. |

**Solution: DFS**

从每个位置开始DFS，直到遍历完所有可能的单词，在这个过程中检查当前单词是否在字典中。注意到每一棵DFS都对应一棵trie的结构，因此我们可以考虑把字典里的词放进一个trie里，在DFS的过程中遍历这课trie。当trie中结点为NULL时，说明字典里没有该前缀，DFS可以提前终止。

|  |
| --- |
| class TrieNode {  public:  // Initialize your data structure here.  TrieNode() {};  TrieNode \*children[26] = {NULL};  ~TrieNode() {  for (int k = 0; k < 26; ++k) delete children[k];  }  bool is\_word = false;  };  **vector<string> findWords(vector<vector<char>>& board, vector<string>& words)** {  //build Trie tree  TrieNode\* root = new TrieNode();  for (string word: words) {  TrieNode \*cur = root;  for (char c : word) {  if (cur->children[c-'a'] == NULL) cur->children[c-'a'] = new TrieNode();  cur = cur->children[c-'a'];  }  cur->is\_word = true;  }    //DFS with early stopping  vector<string> result;  for (int i = 0; i < board.size(); i++) {  for (int j = 0; j < board[0].size(); j++) {  findWords\_recur(board, string(""), i, j, root, result);  }  }    delete root; // delete trie tree  return result;  }  void findWords\_recur(vector<vector<char>>& board, string prefix, int i, int j,  TrieNode \*t, vector<string> &result) {  //termination condition: (1) out of boundary (2) (i, j) has been taken  if (i < 0 || i >= board.size() ||j < 0 || j >= board[0].size()) return;  char c = (board[i][j]);  if (c == '.' || t->children[c-'a'] == NULL) return;    t = t->children[c-'a'];  prefix += c;  if (t->is\_word) {  t->is\_word = false;  result.push\_back(prefix);  }  board[i][j] = '.';  findWords\_recur(board, prefix, i-1, j, t, result);  findWords\_recur(board, prefix, i+1, j, t, result);  findWords\_recur(board, prefix, i, j-1, t, result);  findWords\_recur(board, prefix, i, j+1, t, result);  board[i][j] = c; //restore  } |

# 213. House Robber II

|  |
| --- |
| Note: This is an extension of House Robber.  After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.  Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police. |

**Solution: DP**

考虑两种情况：

1. Rob 第一幢房屋：则不能rob最后一桩，没有其余环形约束。转化为对nums[0...n-2]的house rob
2. 不Rob第一幢房屋：则没有任何环形约束。转化为对nums[1...n-1]的house rob

取两者较大值即可。

|  |
| --- |
| **int rob(vector<int>& nums)** {  if (nums.size() == 0) return 0;  if (nums.size() == 1) return nums[0];  int pre1 = nums[0], last1 = pre1;  int pre2 = 0, last2 = nums[1];  for (int k = 2; k < nums.size(); ++k) {  int cur1 = max(pre1 + nums[k], last1);  int cur2 = max(pre2 + nums[k], last2);  pre1 = last1, pre2 = last2, last1 = cur1, last2 = cur2;  }  return max(pre1, last2);  } |

# 

# 214. Shortest Palindrome

|  |
| --- |
| Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.  For example:  Given "aacecaaa", return "aaacecaaa".  Given "abcd", return "dcbabcd". |

算法1：从字符串中心开始，向左扫描每一个可能的字符串对称轴位置，直到找到回文检查能到达字符串左端的串。

算法2：令字符串为s，它的reverse为rs，我们构造字符串s+#+rs（假设#不在字符串中），则我们可以寻找该字符串的最长匹配前缀和后缀。这由KMP的partial match table构造可以实现。

例如：abcd: abcd#dcba -> 1 -> dcb + abcd=dcbabcd

aacecaaa: aacecaaa#aaacecaa -> 7 -> a+aacecaaa = aaacecaaa

|  |
| --- |
| **string shortestPalindrome(string s)** {  if (s.size() <= 1) return s;    string rs(s.rbegin(), s.rend());  string needle = s + '#' + rs;  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;  }    return rs.substr(0, rs.size()-table.back())+s;  } |

# 

# 215. K-th Largest Element in an Array

|  |
| --- |
| Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.  For example,  Given [3,2,1,5,6,4] and k = 2, return 5.  Note:  You may assume k is always valid, 1 ≤ k ≤ array's length. |

**Solution: pivoting**

使用快速排序的思路。每次以（子）数组首元素为pivot，把数组划分为两部分。如果pivot是sorted后第N-k个元素，则返回pivot。如果pivot更大，则在比pivot小的元素中快排。不然在比pivot大的元素快排。每次规模减少一半。

|  |
| --- |
| int findKthLargest(vector<int>& nums, int k) {  random\_shuffle(nums.begin(), nums.end());    int idx = nums.size() - k;    int first = 0, last = nums.size()-1;  while (first < last) { // partition submatrix between [first, last]  int pivot = nums[first];  int l = first+1, r = last;  for (int j = first+1; l <= r; ++j) {  if (nums[j] <= nums[first]) swap(nums[j], nums[l++]);  else swap(nums[j--], nums[r--]);  }  swap(nums[first], nums[r]);    if (r == idx) return nums[r];  if (r > idx) last = r-1;  else first = r+1;  }  return nums[first];  } |

# 216. Combination Sum III

|  |
| --- |
| Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers. |
| Example 1:  Input: k = 3, n = 7  Output: [[1,2,4]]  Example 2:  Input: k = 3, n = 9  Output:[[1,2,6], [1,3,5], [2,3,4]] |

**Solution: memorized recursion**

递归计算最大数为m,加数总数为k，和为n的结果：

1. n < 1+...+k 或 n > m+m-1+...(m-k+1) 返回
2. n<m->n=m
3. k=1 返回n
4. k>1:递归考虑m包含和不包含在组合里两种情况。

为了避免重复计算，我们把(n, m, k)的结果存进hashmap.考虑到0<=k<=9, 0<=m<=9，可以用n\*100+m\*10+k来编码。

|  |
| --- |
| **vector<vector<int>> combinationSum3(int k, int n)** {  unordered\_map<int, vector<vector<int>>> cache;  int key = combinationSum3\_recur(k, n, min(9, n), cache);  return cache[key];  }  int combinationSum3\_recur(int k, int n, int m,  unordered\_map<int, vector<vector<int>>> &cache) {  int key = n\*100 + k \* 10 + m;    auto it = cache.find(key);  if (it != cache.end()) return key;  if ( (m+m-k+1)\*k/2 < n || (1+k)\*k/2 > n) { //stop recursion: no solution  cache[key] = vector<vector<int>> {};  return key;  }  if ( k == 1) { //stop recursion: find one number below m equal to n  cache[key] = vector<vector<int>> {{n}};  return key;  }  int key1 = combinationSum3\_recur(k, n, min(m-1, n), cache);  vector<vector<int>> result(cache[key1]);  int key2 = combinationSum3\_recur(k-1, n-m, min(m-1, n-m), cache);  for (vector<int> v : cache[key2]) {  result.push\_back(v);  result.back().push\_back(m);  }  cache[key] = result;  return key;  } |

# 217. Contains Duplicates

|  |
| --- |
| Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct. |

**Solution: hashing (unordered\_set)**

用一个unordered\_set来放已经扫描过的数。如果当前数已经出现在set里，则返回true.否则返回false.

|  |
| --- |
| **bool containsDuplicate(vector<int>& nums)** {  unordered\_set<int> history;  for (int v : nums) {  if (history.find(v) == history.end()) history.insert(v);  else return true;  }  return false;  } |

# 

# 218. The Skyline Problem

|  |
| --- |
| A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are given the locations and height of all the buildings as shown on a cityscape photo (Figure A), write a program to output the skyline formed by these buildings collectively (Figure B).    Buildings Skyline Contour  The geometric information of each building is represented by a triplet of integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the left and right edge of the ith building, respectively, and Hi is its height. It is guaranteed that 0 ≤ Li, Ri ≤ INT\_MAX, 0 < Hi ≤ INT\_MAX, and Ri - Li > 0. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.  For instance, the dimensions of all buildings in Figure A are recorded as:  [ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ] .  The output is a list of "key points" (red dots in Figure B) in the format of [ [x1,y1], [x2, y2], [x3, y3], ... ] that uniquely defines a skyline. A key point is the left endpoint of a horizontal line segment. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.  For instance, the skyline in Figure B should be represented as:[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ].  Notes:  The number of buildings in any input list is guaranteed to be in the range [0, 10000].  The input list is already sorted in ascending order by the left x position Li.  The output list must be sorted by the x position.  There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...[2 3], [4 5], [12 7], ...] |

**Solution: line sweeping**

keypoints的x坐标一定是矩形的左右boundary。在某个x，我们考虑所有左边界已经出现但右边界还没有出现的矩形。高度应该是所有height里最大的。

用一个set存储有效矩形的高度。当出现一个左边界时，插入矩形高度，当出现一个右边界时，删除矩形高度。如果存在多条边界在同一个x，处理完所有边界，再确定是否要输出当前的x和高度。

排序的时间是O(nlog(n))，扫描边界时增加、删除结点时间是O(log n)，因此代价是O(nlogn)。总代价是O(nlogn)

|  |
| --- |
| **vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings)** {  vector<pair<int, int>> points;  vector<pair<int, int>> boundary;  multiset<int> heights;  heights.insert(0);//sentinel height    for (vector<int> rect : buildings) {  boundary.push\_back(make\_pair(rect[0], rect[2]));  boundary.push\_back(make\_pair(rect[1], -rect[2]));  }    sort(boundary.begin(), boundary.end(),  [](pair<int, int>p1, pair<int, int>p2) { return p1.first < p2.first;}  );    for (int k = 0, last\_height = 0; k < boundary.size(); last\_height = \*heights.rbegin()) {  int x = boundary[k].first;  for (; k < boundary.size() && boundary[k].first == x; ++k) {  if (boundary[k].second>0) heights.insert(boundary[k].second);  else heights.erase(heights.find(-boundary[k].second));  }  if (last\_height != \*heights.rbegin())  points.push\_back(make\_pair(x, \*heights.rbegin()));  }  return points;  } |

**Solution: divide and conquer**

|  |
| --- |
| **vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings)** {  return buildings.empty()? vector<pair<int, int>>{}:  getSkyline\_recur(buildings, 0, buildings.size()-1);  }  vector<pair<int, int>> getSkyline\_recur(vector<vector<int>>& buildings,  int start, int end) {  if (start<end) {  int mid=start+(end-start)/2;  return merge\_skyline(getSkyline\_recur(buildings, start, mid),  getSkyline\_recur(buildings, mid+1, end));  }  return vector<pair<int, int>>{{buildings[start][0], buildings[start][2]},  {buildings[start][1], 0}};  }  vector<pair<int, int>> merge\_skyline(vector<pair<int, int>> skyline1,  vector<pair<int, int>> skyline2) {  vector<pair<int, int>> result;  auto it2 = skyline2.begin();  int h1 = 0, h2 = 0, curh = 0;  for (auto it1 = skyline1.begin(); it1 != skyline1.end(); ++it1) {  for (; it2 != skyline2.end() && it2->first < it1->first; ++it2) {  h2 = it2->second;  int newh = max(h1, h2);  if (newh != curh) result.emplace\_back(it2->first, curh=newh);  }    if (it1->first == it2->first) h2 = it2++->second;  h1 = it1->second;  int newh = max(h1, h2);  if (newh != curh) result.emplace\_back(it1->first, curh=newh);  }  result.insert(result.end(), it2, skyline2.end());  return result;  } |

# 

# 219. Contains Duplicate II

|  |
| --- |
| Given an array of integers and an integer k, find out whether there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k. |

**Solution: hashing (unordered\_set)**

用unordered\_set维护最近出现的k个数。当扫描到第j个数时：

1. 把nums[j-k]从队列删除。
2. 检查是否nums[j]在集合中
3. 插入nums[j]

|  |
| --- |
| **bool containsNearbyDuplicate(vector<int>& nums, int k)** {  if (k <= 0) return false;  unordered\_set<int> history;  for (int j = 0; j < nums.size(); ++j) {  if (j > k) history.erase(nums[j-k-1]);  if (history.find(nums[j]) == history.end()) history.insert(nums[j]);  else return true;  }  return false;  } |

# 220. Contain Duplicate III

|  |
| --- |
| Given an array of integers, find out whether there are two distinct indices i and j in the array such that the difference between nums[i] and nums[j] is at most t and the difference between i and j is at most k. |

**Solution: set / balanced BST**

用set，而不是unordered\_set存储最近k个数，当扫描nums[j]时：

1. 删除nums[j-k-1]
2. 检查set中是否有[nums[j]-t, nums[j]+t]中的数（通过搜索lowerbound(nums[j]-t))
3. 插入nums[j]

|  |
| --- |
| **bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t)** {  set<int> history;  for (int j = 0; j < nums.size(); ++j) {  if (j > k) history.erase(nums[j-k-1]);  auto it = history.lower\_bound(nums[j]-t);  if (it != history.end() && abs(\*it - nums[j])<= t) return true;  else history.insert(nums[j]);  }  return false;  } |

# 221. Maximal Square

|  |
| --- |
| Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area. |
| For example, given the following matrix:  1 0 1 0 0  1 0 1 1 1  1 1 1 1 1  1 0 0 1 0  Return 4. |

**Solution:**

用2D矩阵M[i][j]存储以(i,j)为右下角的最大正方形边长。H[i][j]存储以(i, j)为右下角的宽为1的最高长方形的高。W[i][j]存储以(i, j)为右下角的高为1的最宽长方形的宽。因此

1. matrix[i][j] = 0: H, M, W = 0
2. matrix[i][j] = 1: H[i, j] = H[i+1, j]+1, W[i, j] = W[i-1, j]+1  
   M[i,j] = max(H[i, j], W[i, j], M[i-1, j-1]+1)

在实现中，我们可以只存最后一行，把空间复杂度减到线性。

|  |
| --- |
| **int maximalSquare(vector<vector<char>>& matrix)** {  if (matrix.size() == 0 || matrix[0].size()==0) return 0;  vector<int> maxw(matrix[0].size(), 0), height(matrix[0].size(), 0);  int max\_area = 0;  for (int i = 0; i < matrix.size(); ++i) {    int width = 0, last = 0;  for (int j = 0; j < matrix[0].size(); ++j) {  int cur = maxw[j];  if (matrix[i][j] == '0') {  height[j] = maxw[j] = width = 0;  }  else {  ++width; ++height[j];  maxw[j] = min(min(width, height[j]), last+1);  max\_area = max(max\_area, maxw[j]\*maxw[j]);  }  last = cur;  }  }  return max\_area;  } |

# 222. Count Complete Tree Nodes

|  |
| --- |
| Given a complete binary tree, count the number of nodes.  Definition of a complete binary tree from Wikipedia:  In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h. |

**Solution: binary search**

通过log(n)的时间可以计算二叉完全树的高度。

给定根结点和它的高度h，则或者左、右子树都为高为h-1的完全树（右子树可能没有填满），或左子树高为h-1且可能没有填满且右子树为填满的高为h-2的完全树。

因此我们只需要递归计算其中一棵子树的结点个数。

|  |
| --- |
| **int countNodes(TreeNode\* root)** {  int h = get\_height(root);  return countNodes\_recur(root, h);  }  int countNodes\_recur(TreeNode\* root, int h) {  if (h <= 1) return h;    int hr = get\_height(root->right);  int count\_left = hr == h-1? (1<< (h-1)) - 1 : countNodes\_recur(root->left, h-1);  int count\_right = hr == h-1? countNodes\_recur(root->right, h-1) : (1<<(h-2)) - 1;  return count\_left + count\_right + 1;  }  int get\_height(TreeNode\* root) {  int count = 0;  for (; root; root = root->left) count++;  return count;  } |

# 223. Rectangle Area

|  |
| --- |
| Find the total area covered by two rectilinear rectangles in a 2D plane.  Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.  Assume that the total area is never beyond the maximum possible value of int.  (A, B)  (C, D)  (E, F)  (G, H) |

**Solution: 2D to 1D separation + interval**

总面积为两个长方形的面积之和减它们交的面积。我们把计算交的面积分解为求两个区间的交的长度，再求其乘积的过程。考虑三种情况：(1)不相交 (2)相交 （3）包含。

|  |
| --- |
| **int computeArea(int A, int B, int C, int D, int E, int F, int G, int H)** {  int a1 = (C - A) \* ( D - B ), a2 = ( G - E ) \* ( H - F );  int a3 = find\_gap(A, C, E, G) \* find\_gap(B, D, F, H);  return a1 + a2 - a3;  }  inline int find\_gap(int l1, int r1, int l2, int r2) {  if (l1 > l2) { swap(l1, l2), swap(r1, r2);}  return (r1 <= l2)? 0 : ((r2<r1? r2:r1) - l2);  } |

# 224. Basic Calculator

|  |
| --- |
| Implement a basic calculator to evaluate a simple expression string.  The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, non-negative integers and empty spaces .  You may assume that the given expression is always valid.  Some examples:  "1 + 1" = 2  " 2-1 + 2 " = 3  "(1+(4+5+2)-3)+(6+8)" = 23 |

**Solution1: stack**

算式可以化简为没有括号的形式。用一个栈维护当前括号的符号，用bool标记上一个运算符。这样扫描完数字串后可以确定当前是加还是减。

|  |
| --- |
| **int calculate1(string s)** {  int sum = 0, num = 0;  stack<bool> signs; signs.push(true);  bool isplus = true;  for (int first = 0; first < s.size(); ++first) {  if (isdigit(s[first])) { //add next number  for (num = 0; isdigit(s[first]); first++) num = num \* 10 + s[first] - '0';  sum += signs.top() == isplus? num : -num;  --first;  }  else if (s[first] != ' '){ //skip spaces  switch (s[first]) { //sign change  case '(': signs.push(isplus==signs.top()); isplus = true; break;  case ')': signs.pop(); break;  default: isplus = s[first]=='+';  }  }  }  return sum;  } |

**Solution2: reverse polish expression**

把算式转化为逆波兰表达式。在碰到运算符和右括号时直接计算一部分值。

转化算法维护一个操作数栈和一个操作符栈。

|  |
| --- |
| **int calculate(string s)** {  stack<int> nums; //operators  stack<char> ops; //operations    s = '(' + s + ')';    for (int k = 0; k < s.size(); ++k) {  if (s[k] == ' ') continue;    // get a number  if (isdigit(s[k])) {  int num = 0;  for (; k < s.size() && isdigit(s[k]); num = num\*10+s[k]-'0', ++k);  --k;  nums.push(num);  }  else if (s[k] == ')') {  while (true) {  char op = ops.top(); ops.pop();  if (op == '(') break;  int n2 = nums.top(); nums.pop();  int n1 = nums.top(); nums.pop();  nums.push(op == '+'? n1+n2:n1-n2);  }  }  else if (s[k] == '(') {  ops.push(s[k]);  }  else {  if (!ops.empty() && ops.top() != '(') {  int n2 = nums.top(); nums.pop();  int n1 = nums.top(); nums.pop();  char op = ops.top(); ops.pop();  nums.push(op == '+'? n1+n2:n1-n2);  }  ops.push(s[k]);  }  }    return nums.top();  } |

# 225. Implement Stack Using Queues

|  |
| --- |
| Implement the following operations of a stack using queues.  push(x) -- Push element x onto stack.  pop() -- Removes the element on top of the stack.  top() -- Get the top element.  empty() -- Return whether the stack is empty.  Notes:  You must use only standard operations of a queue -- which means only push to back, peek/pop from front, size, and is empty operations are valid.  Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.  You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack). |

算法：先考虑如果只有两个元素。为了让queue中的元素顺序颠倒，需要讲队头出栈再进栈。

现在考虑已经有T个元素[T-1, …, 1, 0]，现在T进入队尾，为了T能进入队头，我们把[T-1...0]依次移出队列，加入队尾即可。

|  |
| --- |
| class Stack {  public:  // Push element x onto stack.  void push(int x) {  q.push(x);  for (int k = 0; k < q.size()-1; ++k) {  q.push(q.front());  q.pop();  }  }  // Removes the element on top of the stack.  void pop() {  q.pop();  }  // Get the top element.  int top() {  return q.front();  }  // Return whether the stack is empty.  bool empty() {  return q.empty();  }    private:  queue<int> q;  }; |

# 226. Invert Binary Tree

|  |
| --- |
| Invert a binary tree. |
| 4 4  / \ / \  2 7 => 7 2  / \ / \ / \ / \  1 3 6 9 9 6 3 1 |

**Solution1: recursion**

互换并invert左右子树。

|  |
| --- |
| TreeNode\* invertTree(TreeNode\* root) {  invertTree\_recur(root);  return root;  }  void invertTree\_recur(TreeNode\* root) {  if (root == NULL) return;  swap(root->left, root->right);  invertTree\_recur(root->left);  invertTree\_recur(root->right);  } |

**Solution2: iterative impl. of pre-order traversal**

|  |
| --- |
| TreeNode\* invertTree(TreeNode\* root) {  stack<TreeNode \*> stk;  stk.push(root);    while (!stk.empty()) {  TreeNode \*t = stk.top();  stk.pop();  if (t != NULL) {  swap(t->left, t->right);  stk.push(t->left);  stk.push(t->right);  }  }  return root;  } |

# 227. Basic Calculator II

|  |
| --- |
| Implement a basic calculator to evaluate a simple expression string.  The expression string contains only non-negative integers, +, -, \*, / operators and empty spaces . The integer division should truncate toward zero.  You may assume that the given expression is always valid.  Some examples:  "3+2\*2" = 7  " 3/2 " = 1  " 3+5 / 2 " = 5  Note: Do not use the eval built-in library function. |

**Solution: math ops**

用term维护当前的乘除法项，用total维护当前乘除法项之前各项的和。

term在看到加减号时初始化，乘除号时更新，total只在看到加减号时更新。

|  |
| --- |
| **int calculate(string s)** {  istringstream expression('+' + s + '+');  int total = 0, term = 0;  char op;  while (expression >> op) {  if (op == '+' || op == '-') {  total += term;  if (!(expression >> term)) break;  term = op == '+'? term: -term;  }  else {  int n;  expression >> n;  term = op == '\*'? term\*n : term/n;  }  }  return total;  } |

# 228. Summary Range

|  |
| --- |
| Given a sorted integer array without duplicates, return the summary of its ranges.  For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"]. |

**Solution: simple iteration**

从左到右扫描即可。

|  |
| --- |
| **vector<string> summaryRanges(vector<int>& nums)** {  vector<string> result;  for (int first = 0; first < nums.size(); ) {  int last;  for (last = first+1; last < nums.size() && nums[last] == nums[last-1]+1; last++);  string s = to\_string(nums[first]);  if (first < last-1) {  s += "->" + to\_string(nums[last-1]);  }  result.push\_back(s);  first = last;  }  return result;  } |

# 229. Majority Elements II

|  |
| --- |
| Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times. The algorithm should run in linear time and in O(1) space. |

**Solution: moore voting**

我们试图把数字尽量分配成三个一组，要求每组各不相同。最后剩下的两个值是可能的maority element.

|  |
| --- |
| **vector<int> majorityElement(vector<int>& nums)** {  if (nums.size() <= 1) return nums;    int v1 = nums[0], v2 = nums[0], count1 = 0, count2 = 0;  for (int v : nums) {  if (v == v1) { count1++; }  else if (v == v2) { count2++; }  else if (count1 == 0) { v1 = v; count1 = 1; }  else if (count2 == 0) { v2 = v; count2 = 1; }  else { count1--; count2--; }  if (count1 < count2) {  swap(count1, count2);  swap(v1, v2);  }  }  int c1 = 0, c2 = 0;  for (int v : nums) {  if (v == v1) c1++;  if (v == v2) c2++;  }    vector<int> result;  if (c1 > nums.size()/3) result.push\_back(v1);  if (c2 > nums.size()/3 && v1 != v2) result.push\_back(v2);  return result;  } |

# 230. K-th Smallest Elements in a BST

|  |
| --- |
| Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.  Note:  You may assume k is always valid, 1 ≤ k ≤ BST's total elements.  Follow up:  What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine? |

**Solution: In-order traversal**

如果经常调用，还在node中存储每棵子树的大小。这样所有操作代价都是O(log(n)).

|  |
| --- |
| **int kthSmallest(TreeNode\* root, int k)** {  stack<TreeNode \*> stk;  TreeNode \*t = root;  int count = 0;  while (t != NULL || !stk.empty()) {  if (t != NULL) {  stk.push(t);  t = t->left;  }  else {  t = stk.top();  stk.pop();  ++count;  if (count == k) return t->val;  t = t->right;  }  }  return 0;  } |

# 231. Power of Two

|  |
| --- |
| Given an integer, write a function to determine if it is a power of two. |

**Solution: bit trick**

2的整次幂的二进制形式只有一个1。而操作n&(n-1)把n的最后一位1置0.因此我们通过n&(n-1)==0来判断。(注意0和负数不是power of two)

|  |
| --- |
| **bool isPowerOfTwo(int n)** {  return n > 0 && (n & (n-1))==0;  } |

# 232. Implement Queues using Stack

|  |
| --- |
| Implement the following operations of a queue using stacks.  push(x) -- Push element x to the back of queue.  pop() -- Removes the element from in front of queue.  peek() -- Get the front element.  empty() -- Return whether the queue is empty.  Notes:  You must use only standard operations of a stack -- which means only push to top, peek/pop from top, size, and is empty operations are valid.  You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue). |

**Solution:**

让元素两次进出stack后可以变成先近先出。因此我们使用两个栈，一个栈用于pop和peak，一个栈用于push。在需要pop和peak时，如果输出栈还有元素，则可以直接pop/peak。否则把输入栈中的元素全部倒进输出栈。这样保证了：输入栈的元素晚于输出栈元素，输出栈中的元素是先进先出的。

|  |
| --- |
| class Queue {  public:  // Push element x to the back of queue.  void push(int x) {  stk1.push(x);  }  // Removes the element from in front of queue.  void pop(void) {  if (stk2.empty()) move();  stk2.pop();  }  // Get the front element.  int peek(void) {  if (stk2.empty()) move();  return stk2.top();  }  // Return whether the queue is empty.  bool empty(void) {  return stk1.empty() && stk2.empty();  }  private:  void move() {  while (!stk1.empty()) {  stk2.push(stk1.top());  stk1.pop();  }  }  stack<int> stk1, stk2;  }; |

# 

# 233. Number of Digit One

|  |
| --- |
| Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n. |
| For example:  Given n = 13,  Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13. |

**Solution:**

令n = nk...n1n0. C(n)为1...n中含1的个数，则我们有：

1. nk=0: C(n) = C(nk-1..n1n0).
2. nk=1: C(n) = C(nk-1..n1n0) + [nk-1..n1n0] + 1 + maxk-1.
3. nk>1: C(n) = C(nk-1..n1n0) + (nk-1-1)\*maxk-1+ maxk-1 + 10k  
    = C(nk-1..n1n0) + nk-1\*maxk-1+ 10k
4. C(0) = 0, C(n) = 1, for 1<=n<=9

这里maxk= max C(nk...n1n0) = k\*10k.很容易通过(3)(4)用数学归纳法证明。

因此我们遍历n的每一位，计算它对应的contribution.

|  |
| --- |
| **int countDigitOne(int n)** {  int base = 1, sum = 0, remainder = 0;  for (int k = 0; n>0; ++k, n /= 10, base \*= 10) {  int digit = n % 10;  sum += digit\*k\*(base/10);  if (digit > 0) sum += digit > 1? base : remainder + 1;  remainder = remainder + base \* digit;  }  return sum;  } |

# 234. Palindrome Linked List

|  |
| --- |
| Given a singly linked list, determine if it is a palindrome.  Follow up:  Could you do it in O(n) time and O(1) space? |

**Solution: linked list ops**

把链表对半分为两条（用快慢指针）。第二个链表翻转。然后比较两个链表即可。如果必要，可以恢复输入。

|  |
| --- |
| bool isPalindrome(ListNode\* head) {  for (ListNode \*h2 = reverse(splitHalf(head));  head && h2; head = head->next, h2 = h2->next)  if (head->val != h2->val) return false;  return true;  }  ListNode \*splitHalf(ListNode\* head) {  ListNode sentinel(0); sentinel.next = head;  ListNode \*slow, \*fast;  for (slow = fast = &sentinel; fast; slow = slow->next, fast = fast->next) {  fast = fast->next;  if (!fast) break;  }  ListNode \*retval = slow->next;  slow->next = NULL;  return retval;  }  ListNode \*reverse(ListNode\* head) {  ListNode \*tail = NULL;  while (head) {  ListNode \*next = head->next;  head->next = tail;  tail = head;  head = next;  }  return tail;  } |

# 235. Lowest Common Ancestor of a Binary Search Tree

|  |
| --- |
| Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.  According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).” |
| \_\_\_\_\_\_\_6\_\_\_\_\_\_  / \  \_\_\_2\_\_ \_\_\_8\_\_  / \ / \  0 4 7 9  / \  3 5  For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition. |

**Solution:**

对每个结点，检查是否：

1. p, q属于同一子树：指向子树顶点
2. p或q为顶点，q/p属于某一子树：顶点是LCA
3. p,q属于不同子树：顶点是LCA

|  |
| --- |
| **TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q)** {  while (((root->val-p->val) ^ (root->val- q->val)) > 0 && (root != p) && (root != q))  root = root->val > p->val? root->left : root->right;  return root;  } |

# 236. Lowest Common Ancestor of a Binary Tree

|  |
| --- |
| Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. |
| According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”  \_\_\_\_\_\_\_3\_\_\_\_\_\_  / \  \_\_\_5\_\_ \_\_\_1\_\_  / \ / \  6 2 0 8  / \  7 4  For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition. |

**Solution:**

递归寻找p和q:递归返回子树中p, q个数。它来源于：左子树返回值c1, 右子树返回值c2，顶点的贡献mid(若为p/q，则为1,否则为0）。

对每个结点，检查是否：

1. p, q属于同一子树：指向子树顶点(即c1==2，或c1+mid<2 && c1+mid+c2=2)
2. p或q为顶点，q/p属于某一子树：顶点是LCA
3. p,q属于不同子树：顶点是LCA

|  |
| --- |
| **TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q)** {  TreeNode \*retval = NULL;  LCA\_recur(root, p, q, retval);  return retval;  }  int LCA\_recur(TreeNode\* root, TreeNode\* p, TreeNode\*q, TreeNode \*&retval) {  if (root == NULL) return 0;  int c1 = LCA\_recur(root->left, p, q, retval);  if (c1 == 2) return 2;  int mid = (root ==p || root == q)? 1:0;  if (c1+mid ==2) { retval = root; return 2; }    int c2 = LCA\_recur(root->right, p, q, retval);  if (c1+mid+c2 == 2 && c2 != 2) retval = root;  return c1+mid+c2;  } |

# 237. Delete Node in a Linked List

|  |
| --- |
| Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.  Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function. |

**Solution: brain teaser**

给定当前node，我们因为不知道它的前驱，所以只可能删除它的下一个结点。但我们可以把当前结点的值置为下一个结点，从而“删除”了该结点。

|  |
| --- |
| **void deleteNode(ListNode\* node)** {  ListNode \*next = node->next;  node->val = next->val;  node->next = next->next;  delete next;  } |

# 238. Product of Array Except Self

|  |
| --- |
| Given an array of n integers where n > 1, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].  Solve it without division and in O(n).  For example, given [1,2,3,4], return [24,12,8,6].  Follow up:  Could you solve it with constant space complexity? (Note: The output array does not count as extra space for the purpose of space complexity analysis.) |

**Solution: DP**

为了计算result[k]位置的输出，我们需要计算num[1...k-1]的积和num[k+1...N-1]的积。我们通过两遍扫描可以分别算出这两个积。

|  |
| --- |
| vector<int> productExceptSelf(vector<int>& nums) {  vector<int> result(nums.size(), 0);  for (int k = nums.size()-1, last = 1; k>=0; last \*= nums[k--]) result[k] = last;  for (int k = 0, last = 1; k < nums.size(); last \*= nums[k++]) result[k] \*= last;  return result;  } |

# 239. Sliding Window Maximum

|  |
| --- |
| Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.  Note: You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.  Follow up: Could you solve it in linear time? |
| For example,  Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.  Window position Max  [1 3 -1] -3 5 3 6 7 3  1 [3 -1 -3] 5 3 6 7 3  1 3 [-1 -3 5] 3 6 7 5  1 3 -1 [-3 5 3] 6 7 5  1 3 -1 -3 [5 3 6] 7 6  1 3 -1 -3 5 [3 6 7] 7  Therefore, return the max sliding window as [3,3,5,5,6,7]. |

**Solution1：multiset**

维护一个大小为k的multiset，每次删去过期的数，取最大数。

|  |
| --- |
| **vector<int> maxSlidingWindow(vector<int>& nums, int k)** {  multiset<int> history;  vector<int> result;  for (int j = 0; j < nums.size(); ++j) {  if (j >= k) history.erase(history.find(nums[j-k]));  history.insert(nums[j]);  if (j >= k-1) result.push\_back(\*history.rbegin());  }  return result;  } |

**Solution2:**

用双向链表记录可能成为最大值的数。每当新进一个元素时，窗中小于等于它的数都不可能成为最大值，因此我们pop掉这些数再把当前元素插入。我们观察到队列中的数是单调减、由早到晚排列的。因此我们删除窗口左边元素只需要在必要时pop队头，取最大值即peek队头。

|  |
| --- |
| **vector<int> maxSlidingWindow(vector<int>& nums, int k)** {  vector<int> result;  list<int> max\_pos;  for (int j = 0; j < nums.size(); ++j) {  if (!max\_pos.empty() && (j-max\_pos.front()==k)) max\_pos.pop\_front();  while (!max\_pos.empty() && nums[max\_pos.back()] <= nums[j]) max\_pos.pop\_back();  max\_pos.push\_back(j);  if (j >= k-1) result.push\_back(nums[max\_pos.front()]);  }  return result;  } |

# 

# 

# 240. Search a 2D Matrix II

|  |
| --- |
| 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 in ascending from left to right.  Integers in each column are sorted in ascending from top to bottom. |
| For example, consider the following matrix:  [  [1, 4, 7, 11, 15],  [2, 5, 8, 12, 19],  [3, 6, 9, 16, 22],  [10, 13, 14, 17, 24],  [18, 21, 23, 26, 30]  ]  Given target = 5, return true.  Given target = 20, return false. |

**Solution1：binary search。**

算法复杂度O(mlog(n))

|  |
| --- |
| bool searchMatrix(vector<vector<int>>& matrix, int target) {  for (vector<int> &v : matrix) {  auto it = lower\_bound(v.begin(), v.end(), target);  if (it != v.end() && (\*it) == target) return true;  }  return false;  } |

**Solution2：DP**

在矩阵右上角：如果target小于它则排除最后一列。如果target大于它则排除第一行。这样我们每次把行坐标或列坐标改变1，最坏情况经过m+n步到达左下角。因此复杂度O(m+n)

|  |
| --- |
| bool searchMatrix(vector<vector<int>>& matrix, int target) {  if (matrix.size() == 0 || matrix[0].size() == 0) return false;    int i = 0, j = matrix[0].size()-1;  while (i < matrix.size() && j >=0) {  if (matrix[i][j] == target) return true;  if (matrix[i][j] < target) ++i; else --j;  }  return false;  } |

# 241. Different Ways to Add Parentheses

|  |
| --- |
| Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and \*. |
| Example 1  Input: "2-1-1".  Output: [0, 2]  Because:  ((2-1)-1) = 0  (2-(1-1)) = 2  Example 2  Input: "2\*3-4\*5"  Output: [-34, -14, -10, -10, 10]  Because:  (2\*(3-(4\*5))) = -34  ((2\*3)-(4\*5)) = -14  ((2\*(3-4))\*5) = -10  (2\*((3-4)\*5)) = -10  (((2\*3)-4)\*5) = 10 |

**Solution: memorized recursion**

考虑表达式对应的二叉树。本题相当于给定中序遍历，求所有可能的二叉树。

|  |
| --- |
| **vector<int> diffWaysToCompute(string input)** {  vector<int> operands;  vector<char> ops;    // preprocessing  for (int k = 0; k < input.size(); ++k) {  int num = 0;  for (; k < input.size() && isdigit(input[k]); ++k) num = num \* 10 + input[k] - '0';  operands.push\_back(num);  if (k < input.size()) ops.push\_back(input[k]);  }    // recursion  unordered\_map<long, vector<int>> cache;  diffWaysToCompute\_recur(0, operands.size()-1, operands, ops, cache);  return cache[get\_key(0, operands.size()-1)];  }  void diffWaysToCompute\_recur(int first, int last, vector<int> &operands, vector<char> &ops, unordered\_map<long, vector<int>> &cache) {    long key = get\_key(first, last);  auto it = cache.find(key);  if (it != cache.end()) return; //no need to compute    vector<int> result;  if (first == last) result.push\_back(operands[first]);  else {  for (int k = first; k < last; ++k) {  diffWaysToCompute\_recur(first, k, operands, ops, cache);  diffWaysToCompute\_recur(k+1, last, operands, ops, cache);  vector<int> &s1 = cache[get\_key(first, k)];  vector<int> &s2 = cache[get\_key(k+1, last)];  for (int v1 : s1) {  for (int v2 : s2) {  if (ops[k] == '+') result.push\_back(v1+v2);  if (ops[k] == '-') result.push\_back(v1-v2);  if (ops[k] == '\*') result.push\_back(v1\*v2);  }  }  }  }  cache[key] = result;  }  inline long get\_key(int first, int last) {  return (first << 16) + last;  } |

# 242. Valid Anagram

|  |
| --- |
| Given two strings s and t, write a function to determine if t is an anagram of s.  For example,  s = "anagram", t = "nagaram", return true.  s = "rat", t = "car", return false.  Note:  You may assume the string contains only lowercase alphabets.  Follow up:  What if the inputs contain unicode characters? How would you adapt your solution to such case? |

**Solution: histogram**

计算两个单词的histogram.

|  |
| --- |
| **bool isAnagram(string s, string t)** {  int hist[256] = {0};  for (char c : s) hist[c]++;  for (char c : t) hist[c]--;  for (int k = 0; k < 256; ++k) if (hist[k] != 0) return false;  return true;  } |

# 243. Shortest Word Distance [Locked]

|  |
| --- |
| Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.  Note: You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.  For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].  Given word1 = “coding”, word2 = “practice”, return 3. Given word1 = "makes", word2 = "coding", return 1. |

**Solution: two pointer**

维护指针p1, p2指向word1, word2.每次让较前的指针向后移动到下一位置，并计算两者的距离。

|  |
| --- |
| **int shortestDistance(vector<string>& words, string word1, string word2)** {  int k = 0, j = 0;  while (k < words.size() && word1.compare(words[k]) != 0) ++k;  while (j < words.size() && word2.compare(words[j]) != 0) ++j;  int min\_dist = INT\_MAX;  while (k < words.size() && j < words.size()) {  min\_dist = min(min\_dist, abs(k-j));  if (k < j) {  do {++k;} while (k < words.size() && word1.compare(words[k]) != 0);  }  else {  do {++j;} while (j < words.size() && word2.compare(words[j]) != 0);  }  }  return min\_dist;  } |

# 244. Shortest Word Distance II [Locked]

|  |
| --- |
| This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?  Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.  For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].  Given word1 = “coding”, word2 = “practice”, return 3. Given word1 = "makes", word2 = "coding", return 1.  Note: You may assume that word1 does not equal to word2, and word1 and word2 are both in the list. |

**Solution: hashing (unordered\_map) + merge sort**

把所有word index预存到一个unordered\_map<string, vector<int>>中，然后利用类似merge sort的算法可以得到两个单词的shortest word distance.

|  |
| --- |
| class WordDistance {  public:  WordDistance(vector<string>& words) {  for (int k = 0; k < words.size(); ++k) pos[words[k]].push\_back(k);  }  int shortest(string word1, string word2) {  list<int>& l1 = pos[word1];  list<int>& l2 = pos[word2];  int min\_dist = INT\_MAX;  for (auto it1 = l1.begin(), it2 = l2.begin(); it1 != l1.end() && it2 != l2.end();) {  min\_dist = min(abs(\*it1-\*it2), min\_dist);  if (\*it1 < \*it2) it1++; else it2++;  }  return min\_dist;  }  private:  unordered\_map<string, list<int>> pos;  }; |

# 245. Shortest Word Distance III [Locked]

|  |
| --- |
| This is a follow up of Shortest Word Distance. The only difference is now word1 could be the same as word2.  Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list. word1 and word2 may be the same and they represent two individual words in the list.  Note: You may assume word1 and word2 are both in the list.  For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].  Given word1 = “makes”, word2 = “coding”, return 1.  Given word1 = "makes", word2 = "makes", return 3. |

**Solution: two pointer**

和243题一样，只是如果两个单词相同，则规定前指针移动时不能在后指针处停止。

|  |
| --- |
| **int shortestWordDistance(vector<string>& words, string word1, string word2)** {  int k = 0, j = 0;  while (k < words.size() && word1.compare(words[k]) != 0) ++k;  while (j < words.size() && (j ==k || word2.compare(words[j]) != 0)) ++j;    int min\_dist = INT\_MAX;  while (k < words.size() && j < words.size()) {  min\_dist = min(min\_dist, abs(k-j));  if (k < j) {  do {k++;} while (k < words.size() && (k==j || word1.compare(words[k]) != 0));  }  else {  do {j++;} while (j < words.size() && (k==j || word2.compare(words[j]) != 0));  }  }  return min\_dist;  } |

# 246. Strobogrammatic Number [Locked]

|  |
| --- |
| A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).  Write a function to determine if a number is strobogrammatic. The number is represented as a string.  For example, the numbers "69", "88", and "818" are all strobogrammatic. |

**Solution: front-back pointer**

和回文数一样，只是比较时是把每个数和它的strobogrammatic number比较。

|  |
| --- |
| **bool isStrobogrammatic(string num)** {  int strob[10] = {0, 1, -1, -1, -1, -1, 9, -1, 8, 6};  for (int l = 0, r = num.size()-1; l <= r; l++, r--)  if (num[l]-'0' != strob[num[r]-'0']) return false;  return true;  } |

# 247. Strobogrammatic Number II [Locked]

|  |
| --- |
| A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).  Find all strobogrammatic numbers that are of length = n.  For example,  Given n = 2, return ["11","69","88","96"]. |

**Solution: enumeration**

穷举长为(n+1)/2的所有0, 1, 6, 8, 9组合，构成string的前半部分。后半部分可以通过查表补全。

|  |
| --- |
| **vector<string> findStrobogrammatic(int n)** {  vector<string> result;  string prefix;  findStrobogrammatic\_recur(n, prefix, result);  return result;  }  void findStrobogrammatic\_recur(int n, string &prefix, vector<string> &result){  int strob\_nums[5] = {0, 1, 6, 8, 9};    if (prefix.size() == (n+1)/2) {  int strob[10] = {0, 1, -1, -1, -1, -1, 9, -1, 8, 6};  result.push\_back(prefix);  for (int j = n-(n+1)/2-1; j >=0; --j) result.back()+=(strob[prefix[j]-'0']+'0');  return;  }    prefix += ' ';  for (int k = 0; k < 5; ++k) {  if (1 < n && 1 == prefix.size() && 0 == k) continue;  if (1 == n%2 && (n+1)/2 == prefix.size() && (2 == k || 4 == k)) continue;  prefix.back() = (strob\_nums[k]+'0');  findStrobogrammatic\_recur(n, prefix, result);  }  prefix.pop\_back();  } |

# 248. Strobogrammatic Number III [Locked]

|  |
| --- |
| A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).  Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high. |

**Solution :combinatorial math**

找出所有<low且长度和low一样的strobogrammatic number。

同样，找出所有>high且长度和high一样的strobogrammatic number。

这只需要枚举前一半<low前一半的数量，最后看low前一半是否由strobogrammatic number构成，如果是，则补全这个数并和low比较即可。

如果考虑长度为n的所有strobogrammatic number，一共5(n+1)/2个。

例如：low = 6134

那么长度为4的数字前一半的可能性是1+1\*5 = 7个，分别为：

1001， 1111， 1691， 1881， 1961， 6006，6006

需要注意的是，还要比较一下最后一个数字和low的大小。例如如果low = 6100，那么2112就不应该考虑进来。

|  |
| --- |
| **int strobogrammaticInRange(string low, string high)** {  int nbitl = low.size(), nbith = high.size();  int count = 0;  for (int k = nbitl; k <=nbith; ++k) count += 4\*pow(5, k-1);  count -= stroboLessThan(low);  count -= stroboGreaterThan(high);  if (low == "0") count++;  }  int stroboLessThan(string low) {  int lb[10] = {0, 1, 2, 2, 2, 2, 2, 3, 3, 4};  int strobo[10] = {0, 1, -1, -1, -1, -1, 9, -1, 6, 8};  string last(low);  bool stb\_str = true;  int count = 0;  for (int k = (low.size()+1)/2-1, base = 1; k>=0; --k) {  count += base \* max(lb[low[k]-'0'] - k==0?1:0, 0);  if (strobo[low[k]-'0']<0) stb\_str = false;  last[n-k] = '0'+ strobo[low[n-k]-'0'];  }  if (stb\_str &&  lexicographical\_compare(last.begin(), last.end(), low.begin(), low.end())) count++;  return count;  }  int stroboGreaterThan(string high) {  int ub[10] = {4, 3, 3, 3, 3, 3, 2, 2, 1, 0};  int strobo[10] = {0, 1, -1, -1, -1, -1, 9, -1, 6, 8};  string first(high);  bool stb\_str = true;  int count = 0;  for (int k = (high.size()+1)/2-1, base = 1; k>=0; --k) {  count += base \* ub[high[k]-'0'];  if (strobo[high[k]-'0']<0) stb\_str = false;  first[n-k] = '0'+ strobo[high[n-k]-'0'];  }  if (stb\_str &&  lexicographical\_compare(high.begin(), high.end(), first.begin(), first.end())) {  count++;  }  return count;  } |

# 249. Group Shifted Strings [Locked]

|  |
| --- |
| Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which forms the sequence:  "abc" -> "bcd" -> ... -> "xyz"  Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.  For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],  Return:  [  ["abc","bcd","xyz"],  ["az","ba"],  ["acef"],  ["a","z"]  ]  Note: For the return value, each inner list's elements must follow the lexicographic order. |

**Solution: hashing (unordered\_map)**

用相邻string之间的差距mod26来encode string。并把结果存进hashmap。

具体地：长度为1:0;长度为2：1-26,长度为3: 27-26\*27...因此base为 (pow(26, n-1)-1)/25

|  |
| --- |
| **vector<vector<string>> groupStrings(vector<string>& strings)** {  int nemp = 0;    //assign words to code  unordered\_map<int, vector<string>> grp;  for (string s : strings) {  if (s.size() == 0) ++nemp;  else grp[make\_index(s)].push\_back(s);  }    //produce results and sort lexicographically  vector<vector<string>> result;  if (nemp > 0) result.push\_back(vector<string>(nemp, ""));  for (auto it : grp) {  result.push\_back(it.second);  sort(result.back().begin(), result.back().end());  }  return result;  }  int make\_index(const string &s) {  int start = pow(26, s.size()-1)/25;  int ind = 0;  for (int k = 1; k < s.size(); ++k) {  int diff = (int(s[k])-int(s[k-1]))%26;  ind = ind \* 26 + (diff>=0? 0:26) + diff;  }  return ind + start;  } |

# 250. Count Univalue Subtrees [Locked]

|  |
| --- |
| Given a binary tree, count the number of uni-value subtrees.  A Uni-value subtree means all nodes of the subtree have the same value.  For example:  Given binary tree,  5  / \  1 5  / \ \  5 5 5  return 4. |

**Solution: post-order traversal**

递归返回subtree是否univalue。NULL认为是univalue.

|  |
| --- |
| **int countUnivalSubtrees(TreeNode\* root)** {  int count = 0;  countUnivalSubtrees\_recur(root, count);  return count;  }  bool countUnivalSubtrees\_recur(TreeNode\* root, int &count) {  if (root == NULL) return true;  bool is\_unival\_left = countUnivalSubtrees\_recur(root->left, count);  bool is\_unival\_right = countUnivalSubtrees\_recur(root->right, count);    if ( is\_unival\_left && is\_unival\_right &&  (root->left == NULL || root->left->val == root->val) &&  (root->right == NULL || root->right->val == root->val)) {  ++count;  return true;  }  return false;  } |