# Shortlisted problems

354

370

372

377

# Index

|  |  |  |  |
| --- | --- | --- | --- |
| 351 | recursion | medium |  |
| 352 | iteration | easy | c++ part interesting |
| 353 | list/hashing | easy |  |
| 354 | dp | difficult | classic |
| 355 | hashing | easy |  |
| 356 | sort | easy |  |
| 357 | math | easy |  |
| 358 | bin sort | medium |  |
| 359 | hash | trivial |  |
| 360 | merge sort | easy |  |
| 361 | dp | easy |  |
| 362 | queue | easy |  |
| 363 | map | difficult | very tricky |
| 364 | recursion | easy |  |
| 365 | math | difficult !!! | lot of math |
| 366 | recursion | easy |  |
| 367 | binary search | trivial |  |
| 368 | dp | easy |  |
| 369 | linked list | trivial |  |
| 370 | interval tree; cumsum | difficult |  |
| 371 | bit ops | medium |  |
| 372 | recursion | medium | classic |
| 373 | merge sort | medium |  |
| 374 | binary search | trivial |  |
| 375 | dp | medium |  |
| 376 | iteration | easy |  |
| 377 | enumeration; dp | medium | classic |
| 378 | merge sort | medium |  |
| 379 |  |  |  |
| 380 |  |  |  |
| 381 |  |  |  |
| 382 |  |  |  |
| 383 |  |  |  |
| 384 |  |  |  |
| 385 |  |  |  |
| 386 |  |  |  |
| 387 |  |  |  |
| 388 |  |  |  |
| 389 |  |  |  |
| 390 |  |  |  |
| 391 |  |  |  |
| 392 |  |  |  |
| 393 |  |  |  |
| 394 |  |  |  |
| 395 |  |  |  |
| 396 |  |  |  |
| 397 |  |  |  |
| 398 |  |  |  |
| 399 |  |  |  |
| 400 |  |  |  |

# 351. Android Unlock Patterns [Locked]

|  |
| --- |
| Given an Android 3x3 key lock screen and two integers m and n, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.  Rules for a valid pattern:  Each pattern must connect at least m keys and at most n keys.  All the keys must be distinct.  If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.  The order of keys used matters. |

**Solution1：recursion**

如果不考虑android pattern的约束条件，这个问题就是枚举所有长度为m, m+1, …, n的permutation。考虑andorid pattern时需要加一个额外条件：检查上一位置和当前位置的middle point是否已经访问过。

状态属性：

* visited：长度为9的bit set（实现中用int)，从右数第k位为0表示改数字还没访问过，1表示访问过
* last: 一个0-8之间的数字，上一个move的位置，在android pattern规则中用来排除下一个move。

递归结构：

令C(visited, last)表示(visited, last)出发所有枚举的个数，valid(k, visited, last)表示k未被访问过且不穿过被访问过的数字, #visited表示已访问过的数字长度。

C(visited, last) = sum C(visited | (1 << k), k) + (#visited>=m) over valid(k, visited, last);

初始状态：（4， 0）

visited=0表示没有key被访问过。因为4（即android pattern的5）不排除任何位置，因此初始时last为4.

结束条件：#visited==n时 C(visited, last) = 1.

计算#visited:

我们在递归中维护这个值，从而可以避免反复为不同的states计算。

计算valid(k, visited, last):

1. 通过检查visited可以判断k是否未被访问过: (visited & (1<<k))==0
2. 我们可以从键值key直接计算它所在位置（行、列的index）: (key/3, key%3)  
   对valid而言，键值为k, last
3. 对两个位置(x1, y1), (x2, y2)，可以通过它们和sum\_x=x1+x2, sum\_y=y1+y2的奇偶性判断是否穿过别的键值: sum\_x%2 || sum\_y %2
4. 如果sum\_x和sum\_y都为偶数，则中点为sum\_x/2, sum\_y/2，键值为mid\_point = sum\_x/2\*3+sum+y/2，通过检查visited中的相应位判断： (visited & (1<<mid\_point))

复杂度: 枚举算法是一个指数级算法，call stack因为存了3个数，所以占了线性的空间。但由于call stack的层数不超过9，因此复杂度也可以认为是常数。不考虑android pattern不能穿过未访问键值的规则，m=1, n=9时可能的串有9!个。心算估计这个值的范围在18^4 （略小于大约20^4＝160,000)到30^4（810,000）之间。

|  |
| --- |
| int numberOfPatterns(int m, int n) {//preassumption: 1 <=m <= n <= 9  return numberOfPatterns\_recur(m, n, 0, 0, 4);  }  int numberOfPatterns\_recur(int m, int n, int len, int visited, int last) {  if (n == len) return 1;  int count = (m <= len);    for (int k = 0; k < 9; ++k) {  if (valid(k, visited, last))  count += numberOfPatterns\_recur(m, n, len+1, visited | (1 << k), k);  }  return count;  }  inline bool valid(int cur, int visited, int last) {  if (is\_set(visited, cur)) return false; //check if visited    int sum\_x = last/3 + cur/3;  int sum\_y = last%3 + cur%3;  if (visited == 0 || sum\_x % 2 || sum\_y %2) return true; //check if cross some key    int mid\_point = sum\_x/2\*3 + sum\_y/2;  return is\_set(visited, mid\_point); //check if mid point already set  }  inline bool is\_set(int visited, int digit) {  return visited & (1 << digit);  } |

**Solution2：memorized recursion**

状态编码：state表示的状态和move序列的顺序无关，因此(cur, state)可能被多次访问。通过增加一个cache缓存计算过的(cur, state)状态。

选择容器：由于state用9位encode，因此从右数第10位开始存cur. 可能的取值有29\*9个，大约(4000-5000个）。这些状态放在一个数组里也足够小，所以不必放在一个hashmap里。

|  |
| --- |
| //see above for definition of valid(k, visited, last)  int numberOfPatterns\_recur( int m, int n, int len, int visited, int last,  array<int, 5120>& cache ) {  if (n == len) return 1;    int code = visited + (last << 9); //code key for current state  if (cache[code]>=0) return cache[code];    int count = (m <= len);  for (int k = 0; k < 9; ++k) {  if (valid(k, visited, last))  count += numberOfPatterns\_recur(m, n, len+1, visited | (1 << k), k, cache);  }  cache[code] = count;  return count;  }  int numberOfPatterns(int m, int n) {  array<int, 5120> cache;  cache.fill(-1);  return numberOfPatterns\_recur(max(m, 0), min(n, 9), 0, 0, 4, cache);  } |

**Solution3：DP**

如果问题是求m=1, n=9的特殊情况，则几乎所有的(visited, last)pair都会被访问到。因此我们直接填充一个2D数组count[][]。count[visited][last]表示(visited, last)的可能sequence数。

我们从尾到头扫描visited。由于visited<visited + (1<<k)，因此count[visited+(1<<k)][k]一定已经被填充好了。至于visited里有多少1我们可以用一轮扫描预处理一下。

|  |
| --- |
| inline int compute\_count(int k, int j, array<int, 512\*9>& cache) {  int count = 0;  for (int l = 0; l < 9; ++l)  if (valid(l, k, j)) count += cache[(k | (1<<l))\*9+l];  return count;  }  int numberOfPatterns(int m, int n) {  array<int, 512> ndigit;  ndigit.fill(0);  for (int k = 1; k < 512; ++k) ndigit[k] = ndigit[k & (k-1)] + 1;    array<int, 512\*9> cache;  cache.fill(0);  for (int k = 511; k>=0; --k) {  if (ndigit[k] > n) continue;  for (int j = 0; j < 9; ++j)  cache[k\*9+j] =  (ndigit[k] == n)? 1 : (compute\_count(k, j, cache) + (ndigit[k] >= m? 1: 0));  }  return cache[4];  } |

# 352. Data Stream as Disjoint Intervals

|  |
| --- |
| Given a data stream input of non-negative integers a1, a2, ..., an, ..., summarize the numbers seen so far as a list of disjoint intervals.  Follow up:  What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size? |
| For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:  [1, 1]  [1, 1], [3, 3]  [1, 1], [3, 3], [7, 7]  [1, 3], [7, 7]  [1, 3], [6, 7] |
| Definition for an interval.  struct Interval {  int start;  int end;  Interval() : start(0), end(0) {}  Interval(int s, int e) : start(s), end(e) {}  }; |

**Solution1：set**

用set<Interval> m\_intervals 维护当前的disjoint interval。每次addNum相当于插入(val, val)，但可能需要和1-2个区间合并。每次getIntervals返回当前类里保存的m\_intervals。

addNum:

1. 定义comparator，按end域排序。
2. 寻找需要合并的区间起点：end>=val-1的第一个位置
3. 需要合并的区间终点或为m\_intervals.end()或为第一个start>val+1的位置
4. 对每一个参与合并的区间，刷新合并后区间的起点和终点，然后删除
5. 最后插入(first, last)

getIntervals: 返回m\_intervals

算法复杂度：addNum复杂度是O(log n)的。getIntervals复杂度是O(n)

C++相关：已经存进set的值是read-only的，因此不支持in-place修改。

|  |
| --- |
| **class SummaryRanges {**  **public:**  **SummaryRanges() {}**  **void addNum(int val)** {  auto it = m\_intervals.lower\_bound(Interval(val-1, val-1));  int first = val, last = val;  while (it != m\_intervals.end() && val+1>=it->start ) { //val is in cur interval  first = min(first, it->start);  last = max(last, it->end);  it = m\_intervals.erase(it);  }  m\_intervals.insert(Interval(first, last));  }    **vector<Interval> getIntervals()** {  return vector<Interval>(m\_intervals.begin(), m\_intervals.end());  }    private:  struct cmp {  bool operator()(Interval i1, Interval i2) const {  return i1.end < i2.end;  }  };  set<Interval, cmp> m\_intervals;  **};** |

**Solution 2: mutable set**

如果我们要支持in-place set modification，则需要改变Interval的定义，给start, end加mutable修饰词。

addNum: 首先和算法(1)一样，搜索现有disjoint intervals里第一个可能跟val合并的区间。分5种情况：

1. 没有搜到，或搜到的当前区间不跟val相连：插入[val, val]
2. val属于当前区间：直接返回
3. val=当前区间的start-1:更新当前区间的start为val
4. val=当前区间的end+1：更新当前区间的end
5. 第4步结束后可能后一个区间也跟当前区间相连：更新当前区间的end,删除后一个区间

这个算法时间是O(log n)并且只有在必要时才改变set的结构。

getIntervals:因为leetcode定义好了interval，并且规定返回vector，不得不通过线性时间构造getIntervals的返回值。如果可以直接返回set<mutable\_interval>则getIntervals可以降到O(1)时间。

|  |
| --- |
| **class SummaryRanges {**  **public:**  **SummaryRanges() {}**  **void addNum(int val)** {  auto it = m\_intervals.lower\_bound(mutable\_interval{val-1, val-1});  if (it == m\_intervals.end() || it->start>val+1)  m\_intervals.insert(mutable\_interval{val, val});  else if (it->start==val+1) it->start = val;  else if (it->end==val-1){  it->end = val;  auto next\_it = set<mutable\_interval>::iterator(it);  if (++next\_it != m\_intervals.end() && next\_it->start == val+1) {  it->end = next\_it->end;  m\_intervals.erase(next\_it);  }  }  }  **vector<Interval> getIntervals()** {  vector<Interval> result;  for (auto &i : m\_intervals) result.push\_back(Interval(i.start, i.end));  return result;  }  private:  struct mutable\_interval { mutable int start, end; };  struct cmp {  bool operator()(mutable\_interval i1, mutable\_interval i2) const {  return i1.end < i2.end;  }  };  set<mutable\_interval, cmp> m\_intervals;  }; |

**Solution 3: vector**

用vector<Interval> m\_intervals 维护当前的disjoint interval。使用算法2来addNumber。算法的问题在于发生增删时时间是O(n)的，getIntervals部分可以直接返回m\_intervals。因为C++11把这一步优化为move操作,时间是O(1)。

**Followup：**如果disjoint set数量较少，说明发生了大量算法2中case2-5的情况。且其中case5的情况占少数（否则初期还是会发生大量case1的情况)。此时算法2, 3因为只修改不插入，速度更快。算法3因为vector是序列容器，在getIntervals的步骤更快。

# 353. Design Snake Game [Locked]

|  |
| --- |
| Design a Snake game that is played on a device with screen size = width x height. Play the game online if you are not familiar with the game.   * The snake is initially positioned at the top left corner (0,0) with length = 1 unit. * You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1. * Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake. * When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake. |
| Example: (adapted from Leetcode example to make it work for C++)  int width = 3  int height = 2;  vector<pair<int, int>> food{{1, 2},{0, 1}};  SnakeGame game(width, height, food);  cout<<game.move("R"); // returns 0  cout<<game.move("D"); // returns 0  cout<<game.move("R"); // returns 1  //(Snake eats the first food and right after that, the second food appears at (0,1) )  cout<<game.move("U"); // returns 1  cout<<game.move("L"); // returns 2  cout<<game.move("U"); // returns -1  **S** \_ \_ \_ **S** \_ \_ \_ \_ \_ \_ \_ \_ *F* **S** \_ **S** S  \_ \_ *F* \_ \_ *F* \_ **S** *F* \_ S **S** \_ \_ S \_ \_ S  (R) (D) (R) (U) (L) (U) |

**SnakeGame**

+SnakeGame

+move

**snake**

+snake

+extend\_head

+remove\_tail

+get\_tail

+size

-pixels (list of int pair)

**screen**

**hash\_screen**

+hash\_screen

+add

+remove

+exist

-elements

**food\_iterator**

+screen

+add

+remove

+exist

+in\_bound

+get\_code

-width

-height

+food\_iterator

+check

-foods

-index

**Solution：unordered\_set+doubly linked list**

类设计：见右图

* screen: 内部存一个unordered\_set用于快速检索某个位置是否被snake占据。也可以用bitset，取决于width/height的范围。如果要动态选择，可以用strategy pattern。
* snake: 内部存一个double linked list用于获取蛇的首尾信息。
* food\_iterator: 管理 food

边界条件：

* 蛇已死之后move：这里假设之后的move无效，直接返回-1。我们用bool finished表示游戏结束。
* 食物位置和蛇overlap：题目假设不可能发生。但题目的food是事先给定的，蛇的行为是gamer控制的，因此这个api的设计其实不好。
* 食物吃完后move：这里假设蛇还会继续移动，直到死去。在游戏中可能是蛇吃完最后一个food，游戏就结束，返回food.size()就可以了。

move：

1. 在unordered\_set中删除蛇尾
2. 移动list中的蛇头到新位置，检查当前链表头是否会导致Game Over:有没有撞墙、或已经被蛇占用。
3. 如果当前food位置不为蛇头，删除链表尾，否则food移动到下一位，unordered\_set中添加原来的蛇尾

|  |
| --- |
| **class SnakeGame** {  private:  //sparse/dense representation of pixel occupation  class screen {  private:  const int width, height;    public:  screen(int w, int h) : width(w), height(h) {};  virtual bool exist(pair<int, int> &pos) = 0;  virtual void add(pair<int, int> &pos) = 0;  virtual void remove(pair<int, int> &pos) = 0;  bool in\_bound(pair<int, int> &pos) {  return (pos.first >= 0 && pos.second >=0 &&  pos.first < height && pos.second < width);  }  protected:  inline int get\_code(pair<int, int> pos) {  return pos.first\*width + pos.second;  };  };    //screen impl. using hash set  class hash\_screen : public screen {  private:  unordered\_set<int> elements;  public:  hash\_screen(int w, int h) : screen(w, h) {};  bool exist(pair<int, int> &pos) {  return elements.find(get\_code(pos)) != elements.end();  }  void add(pair<int, int> &pos) { elements.insert(get\_code(pos)); }  void remove(pair<int, int> &pos) { elements.erase(get\_code(pos));}  };  //model of snake trajectory  class snake {  private:  list<pair<int, int>> pixels;  public:  snake(int x, int y) { pixels.emplace\_back(x, y);};  pair<int, int> extend\_head(char dir) {  pixels.push\_front(pixels.front());  switch (dir) {  case 'U': --pixels.front().first; break;  case 'D': ++pixels.front().first; break;  case 'L': --pixels.front().second; break;  case 'R': ++pixels.front().second; break;  }  return pixels.front();  }  void remove\_tail() { pixels.pop\_back(); }  pair<int, int> get\_tail() { return pixels.back(); }  int size() { return pixels.size();}  };  //management of food  class food\_iterator {  private:  const vector<pair<int, int>> foods;  int index;  public:  food\_iterator(const vector<pair<int, int>> &f): foods(f), index(0) {}  bool check(pair<int, int> head) {  pair<int, int> food = index<foods.size()? foods[index] : make\_pair(-1, -1);  if (food.first == head.first && food.second == head.second) {  ++index;  return true;  }  else return false;  }  };  unique\_ptr<snake> ptr\_snake;  unique\_ptr<screen> ptr\_screen;  food\_iterator food;    public:  **SnakeGame(int w, int h, vector<pair<int, int>> f)**: food(f) {  ptr\_snake = unique\_ptr<snake>(new snake(0, 0));  ptr\_screen = unique\_ptr<screen>(new hash\_screen(w, h));  }    **int move(string direction)** {  if (ptr\_snake==nullptr) return -1;  auto tail = ptr\_snake->get\_tail();  ptr\_screen->remove(tail);  auto head = ptr\_snake->extend\_head(direction[0]);  if (!ptr\_screen->in\_bound(head) || ptr\_screen->exist(head)) {    ptr\_snake = nullptr;  ptr\_screen = nullptr;  return -1;  }  ptr\_screen->add(head);    if (food.check(head)) ptr\_screen->add(tail);  else ptr\_snake->remove\_tail();  return ptr\_snake->size()-1;  }  }; |

# 354. Russian Doll Envelopes

|  |
| --- |
| You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.  What is the maximum number of envelopes can you Russian doll? (put one inside other) |
| Example:  Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]). |

**算法：Longest Increasing Subsequence**

如果信封e1可以套在e2里面，则e2高度一定高于e1，因此如果把信封按照高度排序，e1一定在左边。排好序后，russion doll envelop还要求宽度也是从左到右单调减的。

这也就是说求排好序后信封宽度数组的Longest Increasing Subsequence的算法即可。

|  |
| --- |
| int maxEnvelopes(vector<pair<int, int>>& envelopes) {  auto cmp = [](pair<int, int>e1, pair<int, int>e2) {  return e1.first < e2.first || (e1.first == e2.first && e1.second > e2.second);  };  sort(envelopes.begin(), envelopes.end(), cmp);    vector<int> env;  for (pair<int, int> e : envelopes) {  auto it = lower\_bound(env.begin(), env.end(), e.second);  if (it == env.end()) env.push\_back(e.second); else \*it = e.second;  }    return env.size();  } |

**Followup：what if the envelopes can be rotated by 90 degree**

因为两个一样大的信封旋转其中之一以后不能envelop each other，所以我们只要把所有信封的旋转也放进LIS的输入就好了。

# 355. Design Twitter

|  |
| --- |
| Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:   * postTweet(userId, tweetId): Compose a new tweet. * getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent. * follow(followerId, followeeId): Follower follows a followee. * unfollow(followerId, followeeId): Follower unfollows a followee. |
| // Example  Twitter twitter;  twitter.postTweet(1, 5); // User 1 posts a new tweet (id = 5).  twitter.getNewsFeed(1); // User 1's news feed should return a list with  // 1 tweet id -> [5].  twitter.follow(1, 2); // User 1 follows user 2.  twitter.postTweet(2, 6); // User 2 posts a new tweet (id = 6).  twitter.getNewsFeed(1); // User 1's news feed should return a list with  // 2 tweet ids -> [6, 5].  // Tweet id 6 should precede tweet id 5 because  // it is posted after tweet id 5.  twitter.unfollow(1, 2); // User 1 unfollows user 2.  twitter.getNewsFeed(1); // User 1's news feed should return a list with  // 1 tweet id -> [5],  // since user 1 is no longer following user 2. |

**Solution**：unordered\_map/unordered\_set + merge sort

数据结构：一个unordered\_map存储从id->follower/tweet的映射。对每个用户纪录

* unordered\_set<int>: 这个id follow的id
* list<pair<int, int>>: 这个id的最近10篇发文，其中pair<int, int>纪录tweet id和time stamp。
* int: current time stamp

(感觉int不是user id和time stamp最好的选择，似乎uint64要好些）

边界条件：

* id不能follow自己（或者by default，id必须follow自己）。
* id必须follow已经post过tweet的id，或者getNewsFeed过的id。
* 需问清实现需不需要考虑线程安全（leetcode上似乎不用，但实际应用中是需要的）

postTweet：当前tweet加入user id最近发文list，如果超出容量须删除最早tweet

getNewsFeed：取user id所有followee和id自己的发文list。放在一个priority queue里进行k-way merge

Follow： 把followee id加入follower的followee list

Unfollow： 把followee id从follower的followee list删掉。如果假设follower必须包括自己则这里不能删自己。followee id不存在于list中的情况由unordered\_set处理。

|  |
| --- |
| struct Compare {  bool operator () (list<pair<int, int>> l1, list<pair<int, int>> l2) {  return l1.back().first < l2.back().first;  }  };  **class Twitter** {  unordered\_map<int, list<pair<int, int>>> mPosts;  unordered\_map<int, unordered\_set<int>> mFollowees;  int mT = 0;  **public:**  **Twitter()** {}    **void postTweet(int userId, int tweetId)** {  list<pair<int, int>> &posts = mPosts[userId];  posts.push\_back(make\_pair(mT++, tweetId));  if (posts.size() > 10) posts.pop\_front();  }    **vector<int> getNewsFeed(int userId)** {  vector<int> result;  priority\_queue<list<pair<int, int>>, vector<list<pair<int, int>>>, Compare> heap;    for (int v : mFollowees[userId]) {  auto it = mPosts.find(v);  if (it != mPosts.end() && it->second.size() > 0) heap.push(it->second);  }  auto it = mPosts.find(userId);  if (it != mPosts.end() && it->second.size() > 0) heap.push(it->second);    for (int k = 0; k < 10 && heap.size()>0; ++k) {  list<pair<int, int>> ltop = heap.top();  heap.pop();  result.push\_back(ltop.back().second);  ltop.pop\_back();  if (ltop.size() > 0) heap.push(ltop);  }  return result;  }    **void follow(int followerId, int followeeId)** {  if (followeeId == followerId) return;  unordered\_set<int> &followees = mFollowees[followerId];  if (followees.find(followeeId) == followees.end()) followees.insert(followeeId);  }    **void unfollow(int followerId, int followeeId)** {  auto it = mFollowees.find(followerId);  if (it == mFollowees.end()) return;  it->second.erase(followeeId);  }  }; |

# 

# 

# 356. Line Reflection [Locked]

|  |
| --- |
| Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given set of points.  Follow up: Could you do better than O(n2)? |
| Example 1: Given points = [[1,1],[-1,1]], return true.  Example 2: Given points = [[1,1],[-1,-1]], return false. |
| Hint:  Find the smallest and largest x-value for all points.  If there is a line then it should be at y = (minX + maxX) / 2.  For each point, make sure that it has a reflected point in the opposite side. |

**Solution: partition**

算法：

1. 取对称轴。如果符合line reflection的话，对称轴为y=(minx+maxx)/2。
2. 把所有点作3way partition。
3. 把第一个partition按先y后x/从小到大sort。第三个partition的y改为minX+maxX-y按从小到大sort.
4. 比较第一个和第三个partition，跳过重复点。

边界条件：

* reflected point可否是自己。如果不允许则还要检查对称轴上各点的频率是否是偶数。
* points为空时认为是true/false?
* 重复点怎么处理

|  |
| --- |
| bool isReflected(vector<pair<int, int>>& points) {  if (points.size() == 0) return true; //need to clarify    int miny = points[0].first, maxy = miny;  for (auto p : points) {  miny = min(miny, p.first);  maxy = max(maxy, p.first);  }  int sumy = maxy + miny;  vector<pair<int, int>> l1, l2;  for (auto p : points) {  if (p.first \* 2 < sumy) l1.push\_back(p);  if (p.first \* 2 > sumy) l2.emplace\_back(sumy - p.first, p.second);  }  sort(l1.begin(), l1.end());  sort(l2.begin(), l2.end());  int k, j;  for (k = 0, j = 0; k < l1.size() && j < l2.size();) {  int x = l1[k].first, y = l1[k].second;  if (l1[k].first != l2[j].first || l1[k].second != l2[j].second) return false;  while (k < l1.size() && l1[k].first==x && l1[k].second==y) ++k;  while (j < l2.size() && l2[j].first==x && l2[j].second==y) ++j;  }  return (k == l1.size() && j == l2.size());  } |

# 357. Count Numbers with Unique Digits

|  |
| --- |
| Given a non-negative integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n. |
| Example:  Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99]) |

**Solution1：combinatorics math**

观察: n位数中unique digit numbers 个个数：

0位数：1 （0）  
1位数：9

2位数：9\*9

3位数：9\*9\*8

…

n+1位数：count[n-1] + 9\*9\*8\*...\*(10-n)

规律:第一位取非0的9种可能性，第k位取和前k-1位不一样的数字(10-k种）

边界条件: n=0: return 1

|  |
| --- |
| **int countNumbersWithUniqueDigits(int n)** {  if (n==0) return 1;  result[0] = 10;  for (int i = 1, base = 9, prod = 9; i < min(10, n); ++i) {  prod \*= (10-i);  result[i] = result[i-1] + prod;  }  return result[n-1];  } |

**Solution2：caching**

因为有意义的输入一共只有10个，我们可以把0-10的结果预存起来，以便快速lookup.

|  |
| --- |
| **int countNumbersWithUniqueDigits(int n)** {  static int result[11] = {1, 10, 91, 739, 5275, 32491, 168571,  712891, 2345851, 5611771, 8877691};  return n>10? result[10] : result[n];  } |

# 358. Rearrange String k Distance Apart [Locked]

|  |
| --- |
| Given a non-empty string str and an integer k, rearrange the string such that the same characters are at least distance k from each other.  All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".  Example 1: str = "aabbcc", k = 3  Result: "abcabc"  The same letters are at least distance 3 from each other.  Example 2: str = "aaabc", k = 3  Answer: ""  It is not possible to rearrange the string.  Example 3: str = "aaadbbcc", k = 2  Answer: "abacabcd"  Another possible answer is: "abcabcda"  The same letters are at least distance 2 from each other. |

**Solution：Radix sort**

算法思想：因为题意是需要rearrange，所以输入的次序不重要，字频才是重要的。假设字频是f1>＝f2>＝f3>＝...>＝fn，对应频率是c1, c2, ...cn。很明显的是字符串长度的lower\_bound是(f1-1)\*k+1，即c1间的空隙(下图实线框）要尽量多填不是c1的字母，但对词频＝f1的其它数字，因为必须放进不同的空隙里，所以最后一个字母可以放在两侧，这里我们统一假设放在虚线框里。

c1

c1

c1

c1

对频率<f1的字母，只需要塞实线框，但我们希望塞得尽量均匀以保证c1间最小距离最大。因此考虑把这几个bin当作rotating array，依次塞进去。

边界条件：

可能有先塞的字母相对位置较前，后塞的字母相对位置较后一位的问题。如下图中的第一行，可能导致塞进的第一个字母和最后一个字母间距较小。

有两种处理办法：

(1) 频率>=f1-1的字母优先入bin

(2) rotating入bin的顺序从后向前。（如第二行）

c1

c1

c1

c1

c

c

c

c1

c1

c1

c1

c

c

c

有解条件：字符串长度-频率为最高频字母数 >= 最高频率f1 \* k

复杂度分析：时间空间都是O(n)的。

|  |
| --- |
| **string rearrangeString(string str, int k)** {  if (str.size()==0) return str;    // get the char with highest frequency  char max\_char; int max\_count = -1;  int counts[256] = {0};  for (char c : str) {  if (++counts[c] > max\_count) {  max\_count = counts[c];  max\_char = c;  }  }    vector<string> bins(max\_count, "");  for (int c = 0; c < 256; ++c) {  if (counts[c]==max\_count) {  for (int j = 0; j < counts[c]; ++j) bins[j] += c;  }  }  for (int c = 0; c < 256; ++c) {  if (counts[c]==max\_count-1) {  for (int j = 0; j < counts[c]; ++j) bins[j] += c;  }  }  int bid = 0;  for (int c = 0; c < 256; ++c) {  if (counts[c]<max\_count-1) {  for (int j = 0; j < counts[c]; ++j) bins[(bid++)%(max\_count-1)] += c;  }  }    string result;  for (int j = 0; j < max\_count; ++j) {  if (j < max\_count-1 && bins[j].size() < k) return "";  result += bins[j];  }  return result;  } |

# 359. Logger Rate Limiter [Locked]

|  |
| --- |
| Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is not printed in the last 10 seconds.  Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.  It is possible that several messages arrive roughly at the same time. |
| Example:  Logger logger;  logger.shouldPrintMessage(1, "foo"); // logging string "foo" at timestamp 1 returns true;  logger.shouldPrintMessage(2,"bar"); // logging string "bar" at timestamp 2 returns true;  logger.shouldPrintMessage(3,"foo"); // logging string "foo" at timestamp 3 returns false;  logger.shouldPrintMessage(8,"bar"); // logging string "bar" at timestamp 8 returns false;  logger.shouldPrintMessage(10,"foo"); // logging string "foo" at timestamp 10 returns false;  logger.shouldPrintMessage(11,"foo"); // logging string "foo" at timestamp 11 returns true; |

**Solution: unordered\_map**

数据结构：unordered\_map<string, int>存储每个logger上一次print的时间。

ShouldPrintMessage：

(1)检索是否有以当前message为key的item.

(2)如有，检查当前是否打印

(3)如打印，更新上次打印时间为现在

|  |
| --- |
| **class Logger** {  public:  **bool shouldPrintMessage(int t, string message)** {  auto it = last.find(message);  if (it != last.end() && it->second + interval > t) return false;  last[message] = t;  return true;  }  private:  const int interval = 10;  unordered\_map<string, int> last;  }; |

# 360. Sort Transformed Array [Locked]

|  |
| --- |
| Given a sorted array of integers nums and integer values a, b and c. Apply a function of the form f(x) = ax2 + bx + c to each element x in the array.  The returned array must be in sorted order.  Expected time complexity: O(n) |
| Example:  nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5,  Result: [3, 9, 15, 33]  nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5  Result: [-23, -5, 1, 7] |

**Solution：merge sort**

算法：

二次函数是一个bitonic结构，转折点在b/2a。因此我们把数据中心找到，reverse其中一半，再in-place归并就好

边界条件：

如nums为空则直接返回本身

a=0: 则array已经排好序或是逆序，(b<0时需要reverse)

a!=0:由于a的符号有可能出现逆序，一种方法是根据a的符号决定reverse一半时选哪一半，另一种是总是reverse某一半，但最后检查是否输出是逆序，如果是再reverse一次。

|  |
| --- |
| **vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c)** {  return (a == 0) ? sortLinearArray(nums, b, c) : sortBitonicArray(nums, a, b, c);  }  vector<int> sortLinearArray(vector<int>& nums, int b, int c) {  vector<int> result(nums);  if (b < 0) reverse(result.begin(), result.end());  for (int& v: result) v = v\*b + c;  return result;  }  vector<int> sortBitonicArray(vector<int>& nums, int a, int b, int c) {  vector<int> numt(nums);  auto it\_mid = lower\_bound(numt.begin(), numt.end(), -b/a/2);  for (int& v: numt) v = v\*v\*a + v\*b + c;  if (a > 0) reverse(numt.begin(), it\_mid); else reverse(it\_mid, numt.end());  vector<int> result;  auto it2 = it\_mid;  for (auto it1 = numt.begin(); it1 != it\_mid; ++it1) {  for (; it2 != numt.end() && \*it2 < \*it1; it2++) result.push\_back(\*it2);  result.push\_back(\*it1);  }  for (; it2 != numt.end(); it2++) result.push\_back(\*it2);  return result;  } |

# 361. Bomb Enemy [Locked]

|  |
| --- |
| Given a 2D grid, each cell is either a wall 'Y', an enemy 'X' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb.  The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.  Note that you can only put the bomb at an empty cell. |
| Example: For the given grid  0 X 0 0  X 0 Y X  0 X 0 0  return 3. (Placing a bomb at (1,1) kills 3 enemies) |

**Solution：Dimension Reduction + DP**

在每个位置放炸弹可以炸掉同行的X或同列的X是独立的。只是如果当前位置有enemy，被数了两次，最后要－1。

对每行可炸的bomb，从左到右扫一次可以知道被墙壁隔开分段的enemy数，从右到左扫一次可以把结果纪录下来。例如对example第二行

Grid row: E 0 W E

L->R scan: 1 2 0 1

R->L scan: 2 2 0 1

类似地对每列也可以这样计算。

扫描列的最后一遍的之后可以计算每个位置的结果，返回最大值。

|  |
| --- |
| int maxKilledEnemies(vector<vector<char>>& grid) {  int M = grid.size(), N = M>0?grid[0].size():0;  if (N == 0) return 0;    vector<vector<int>> dist1(M, vector<int>(N, 0)), dist2(M, vector<int>(N, 0));  for (int i = 0; i < M; ++i) {  for (int j = 0; j < N; ++j) {  if (grid[i][j] == 'W') dist1[i][j] = 0;  else dist1[i][j] = (j == 0? 0 : dist1[i][j-1]) + (grid[i][j] == 'E'? 1:0);  }  for (int j = N-2; j >= 0; j--)  dist1[i][j] = grid[i][j]=='W'? 0:max(dist1[i][j], dist1[i][j+1]);  }  for (int j = 0; j < N; ++j) {  for (int i = 0; i < M; ++i) {  if (grid[i][j] == 'W') dist2[i][j] = 0;  else dist2[i][j] = (i == 0? 0 : dist2[i-1][j]) + (grid[i][j] == 'E'? 1:0);  }  for (int i = M-2; i >= 0; i--)  dist2[i][j] = grid[i][j]=='W'? 0:max(dist2[i][j], dist2[i+1][j]);  }  int result = 0;  for (int j = 0; j < N; ++j)  for (int i = 0; i < M; ++i)  if (grid[i][j] == '0') result = max(result, dist1[i][j] + dist2[i][j]);  return result;  } |

# 362. Design Hit Counter [Locked]

|  |
| --- |
| Design a hit counter which counts the number of hits received in the past 5 minutes.  Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.  It is possible that several hits arrive roughly at the same time.  Follow up:  What if the number of hits per second could be very large? Does your design scale? |
| Example:  HitCounter counter;  counter.hit(1); // hit at timestamp 1.  counter.hit(2); // hit at timestamp 2.  counter.hit(3); // hit at timestamp 3.  counter.getHits(4); // get hits at timestamp 4, should return 3.  counter.hit(300); // hit at timestamp 300.  counter.getHits(300); // get hits at timestamp 300, should return 4.  counter.getHits(301); // get hits at timestamp 301, should return 3. |

**Solution：queue + input aggregation**

用一个queue存储最近300秒（5分钟）的(time stamp,count) pair，返回sum of count。

|  |
| --- |
| class HitCounter {  public:  void hit(int t){  ++nhit;  if (history.empty() || history.back().first < t) history.emplace(t, 0);  history.back().second++;  pop\_history(t);  }    int getHits(int t){  pop\_history(t);  return nhit;  }  private:  void pop\_history(int t) {  while (!history.empty() && history.front().first + 300 <= t) {  nhit -= history.front().second;  history.pop();  }  }  queue<pair<int, int>> history;  int nhit = 0;  }; |

# 363. Max Sum of Rectangle No Larger Than K

|  |
| --- |
| Given a non-empty 2D matrix matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.  Note:  The rectangle inside the matrix must have an area > 0.  What if the number of rows is much larger than the number of columns? |
| Example:  Given matrix = [  [1, 0, 1],  [0, -2, 3]  ]  k = 2  The answer is 2. Because the sum of rectangle [[0, 1], [-2, 3]] is 2 and 2 is the max number no larger than k (k = 2). |

**Solution：cumsum+map**

1D版本：求1D array不大于k的最大子数组。我们先令C[k] = sum A[0...k]，则sum A[i,...j] = C[j]-C[i-1]。

对每个i，用一个map维护所有C[j]:j>i，并搜索C[j]<=C[i-1]+k 的最大C[j]：这可以通过寻找lower\_bound(C[i-1]+k+1)的前一个元素得到.

2D版本：穷举rectangle的左边和右边的所有可能index，从而知道每行的和。这样在循环内部，我们只需要解上面的1D问题即可。

|  |
| --- |
| **int maxSumSubmatrix(vector<vector<int>>& matrix, int k)** {  vector<vector<int>> cumsum(matrix.size(), vector<int>(matrix[0].size()+1, 0));  for (int i = 0; i < matrix.size(); ++i)  for (int j = 0; j < matrix[0].size(); ++j)  cumsum[i][j+1] = cumsum[i][j] + matrix[i][j];    int maxsum = INT\_MIN;  for (int i = 0; i < cumsum[0].size(); ++i) {  for (int j = i+1; j < cumsum[0].size(); ++j) {  set<int> cs;    int sum = 0;  for (int l = 0; l < cumsum.size(); ++l) sum += cumsum[l][j] - cumsum[l][i];  if (sum <= k) maxsum = max(maxsum, sum);  for (int l = cumsum.size()-1; l >= 0; --l) {  cs.insert(sum);  sum -= cumsum[l][j] - cumsum[l][i];  auto it = cs.lower\_bound(sum + k + 1);  if (it != cs.begin()) maxsum = max(maxsum, \*(--it) - sum);  }  }  }  return maxsum;  } |

# 364. Nested List Weight Sum II [Locked]

|  |
| --- |
| Given a nested list of integers, return the sum of all integers in the list weighted by their depth.  Each element is either an integer, or a list -- whose elements may also be integers or other lists.  Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.  Example 1:  Given the list [[1,1],2,[1,1]], return 8. (four 1's at depth 1, one 2 at depth 2)  Example 2:  Given the list [1,[4,[6]]], return 17. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1\*3 + 4\*2 + 6\*1 = 17) |

**Solution: recursion**

由于depth是自底向上的，因此在遍历时也返回深度信息。

|  |
| --- |
| int depthSumInverse(vector<NestedInteger>& nested\_list) {  int depth = get\_depth(nested\_list);  return depthSumInverse\_recur(nested\_list, depth);  }  int get\_depth(vector<NestedInteger>& nested\_list) {  int depth = 1;  for (NestedInteger& ni : nested\_list) {  if (!ni.isInteger()) depth = max(get\_depth(ni.getList())+1, depth);  }  return depth;  }  int depthSumInverse\_recur(vector<NestedInteger>& nested\_list, int depth) {  int sum = 0;  for (NestedInteger& ni : nested\_list) {  if (ni.isInteger()) sum += ni.getInteger() \* depth;  else {  sum += depthSumInverse\_recur(ni.getList(), depth-1);  }  }  return sum;  } |

# 365. Water and Jug Problem

|  |
| --- |
| You are given two jugs with capacities x and y litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly z litres using these two jugs.  If z liters of water is measurable, you must have z liters of water contained within one or both buckets by the end.  Operations allowed:   * Fill any of the jugs completely with water. * Empty any of the jugs. * Pour water from one jug into another till the other jug is completely full or the first jug itself is empty. |
| Example 1: Input: x = 3, y = 5, z = 4 Output: True (From the famous "Die Hard" example)  Example 2: Input: x = 2, y = 6, z = 5 Output: False |

**Solution 1：DAG traversal by BFS**

状态空间：首先如果z<0或z>x+y，则一定不可能。jug里的水量共有(x+1)(y+1)种可能性。我们用状态(a, b)表示jug1有a 升水，jug2有b升水。

状态转移：(a, b) 可能的下一个状态有：

(x, b), (a, y), (0, b), (a, 0), (a+min(b, x-a), b-min(b, x-a)), (a-min(a, y-b), b+min(a, y-b))

初始状态：(0, 0).

因为状态转移里可能有循环，因此用unordered\_set纪录哪些状态被访问过。

复杂度：复杂度是O(xy)，因此在规模较大的x,y上会TLE/MLE.

|  |
| --- |
| **bool canMeasureWater(int x, int y, int z)** {  unordered\_set<int> history;  vector<int> pre;  pre.push\_back(0);  history.insert(0);    while (!pre.empty()) {  vector<int> cur;  for (int code : pre) {  pair<int, int> p = decode(code, y);  if (p.first + p.second == z) return true;    if (x != p.first) insert(cur, history, encode(x, p.second, y));  if (y != p.second) insert(cur, history, encode(p.first, y, y));  if (p.first != 0) insert(cur, history, encode(0, p.second, y));  if (p.second != 0) insert(cur, history, encode(p.first, 0, y));  int q12 = min(p.first, y-p.second);  if (q12 != 0) insert(cur, history, encode(p.first-q12, p.second+q12, y));  int q21 = min(x-p.first, p.second);  if (q21 != 0) insert(cur, history, encode(p.first+q21, p.second-q21, y));  }  swap(cur, pre);  }  return false;  }  inline void insert(vector<int>& cur, unordered\_set<int>& history, int code) {  if (history.find(code) != history.end()) return;  history.insert(code);  cur.push\_back(code);  }  inline int encode(int p1, int p2, int y) {  return p1\*(y+1) + p2;  }  inline pair<int, int> decode(int code, int y) {  return make\_pair(code/(y+1), code % (y+1));  } |

**Solution 2：bin sort**

问题简化：

假如x, y不互质，且最大公约数为a，则我们可以通过把容量单位除以a来得到所有可能的状态。因此当z不能整除a时，答案必为false。如果x=1,则我们只需要反复地灌满x，再把水倒进y里，就可以得到[0, x+y]范围内的任意值。

观察规律：我们只考虑x, y互质的情况。

* 我们一定能measure x的任意倍数。这启发我们可能解和z%x的值有关。

不失一般性，假设y>=x，y=nx+q (0<q<x)：

* 通过(0, y)->(x, y-x)->(0, y-x)->(x, y-2x)->...可以量出(0，q)。

通过反复灌jug1，再从jug1灌到jug2，也能measure nx+q(0<nx+q<=x+y).到此进一步验证了前面的猜想。

* 从(0, q)出发，唯一可能可以改变余数的路径是：  
  (0,q)->(q,0)->(q,y)->(x, y+q-x)->(0, y+q-x)->(x, y+q-2x)->... 直到量出(0, 2q%x)
* 从r=2q%x出发唯一可能可以改变余数的路径是：  
  (0, r)->(r, 0), (r, y)->(x, y-x+r)->(0, y-x+r)->(x, y-2x+r)->...直到量出(0, (q+r)%x)=(0, 3q%x)

从而我们可以量出任意的(bq%x)+nx的刻度，

因此我们只需要考虑0, q, 2q, 3q, …(x-1)q 对x取模是否覆盖[0, x)范围内所有数。这用一个简单的bitset就可以验证。

|  |
| --- |
| bool canMeasureWater(int x, int y, int z) {  int a = \_\_gcd(x, y);  return z == 0 ||  z <= (long)(x+y) && (z%a==0? canMeasureWater\_coprime(x/a, y/a, z/a) : false);  }  bool canMeasureWater\_coprime(int x, int y, int z) {  int min\_xy = min(x, y), q = (x+y)%min\_xy;    vector<bool> hit(min\_xy, false);  for (int k = 0; k < min\_xy; ++k) {  if (hit[k]) return false; else hit[k] = true;  }  return true;  } |

**Solution 3：反证法。**

现在我们进一步考虑canMeasureWater\_coprime的输出，通过实验一些例子，发现都是返回true。

因此考虑反证canMeasureWater\_coprime恒为true。假如输出为false，则：(0，q%x, 2q%x, … (x-1)q%x)没有覆盖[0, x-1]范围内所有的数。因此存在0<=i<j<=x-1，iq %x = jq %x，即(j-i)\*q %x=0.

因为1<=j-i<=x-1，因此q 与x不互质，假设最大公约束为c。 则y = nx+q也整除c，这与x, y互质矛盾。

|  |
| --- |
| bool canMeasureWater(int x, int y, int z) {  return z == 0 || z <= (long)(x + y) && z % \_\_gcd(x, y) == 0;  } |

# 366. Find Leaves of Binary Tree [Locked]

|  |
| --- |
| Given a binary tree, find all leaves and then remove those leaves. Then repeat the previous steps until the tree is empty. |
| Example:  Given binary tree  1  / \  2 3  / \  4 5  Returns [4, 5, 3], [2], [1].  Explanation:  1. Remove the leaves [4, 5, 3] from the tree  1  /  2  2. Remove the leaf [2] from the tree  1  3. Remove the leaf [1] from the tree  []  Returns [4, 5, 3], [2], [1]. |

**Solution1：post-order traversal**

返回自己是不是叶子。如果左右子树是叶子，则删除。

**Solution2：post-order traversal + vector**

只是返回traversal结果，但不执行删除操作。用upside down depth决定自己应该被插到哪一层。

|  |
| --- |
| **vector<vector<int>> findLeaves(TreeNode\* root)** {  vector<vector<int>> result;  findLeaves\_recur(root, result);  return result;  }  int findLeaves\_recur(TreeNode\* root, vector<vector<int>>& result) {  if (root == NULL) return -1;    int depth = findLeaves\_recur(root->left, result) + 1;  depth = max(findLeaves\_recur(root->right, result) + 1, depth);    if (result.size() == depth) result.push\_back(vector<int>{});  result[depth].push\_back(root->val);  return depth;  } |

# 367. Valid Perfect Square

|  |
| --- |
| Given a positive integer num, write a function which returns True if num is a perfect square else False.  Note: Do not use any built-in library function such as sqrt. |
| Examples: 16->True, 14->False |

**Solution：binary searach**

首先用二分法计算num的平方根，然后检查平方根的平方是否为num即可

|  |
| --- |
| **bool isPerfectSquare(int num)** {  int left = 1, right = num;  while (left < right-1) {  int mid = (left + right)/2;  if (mid\*mid == num) return true;  if (mid > num/mid) right = mid-1; else left = mid+1;  }  return left\*left == num || right\*right == num;  } |

# 368. Largest Divisible Subset

|  |
| --- |
| Given a set of distinct positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0.  If there are multiple solutions, return any subset is fine. |
| Example 1: nums: [1,2,3] Result: [1,2] (of course, [1,3] will also be ok)  Example 2: nums: [1,2,4,8] Result: [1,2,4,8] |

**Solution: dynamic programmng**

用C[n]存储以A[n]为结尾的最长sequence的长度，Prev[n]存储A[n]在最长sequence里的上一个单词。则通过动态规划

C[n] = maxA[n]%A[m]==0 C[m]+1

Prev[n] = argmaxm:A[n]%A[m]==0 C[m]

可以得到最长sequence。通过backtrack Prev[]数组可以找到序列。

|  |
| --- |
| **vector<int> largestDivisibleSubset(vector<int>& nums)** {  sort(nums.begin(), nums.end());  vector<int> count(nums.size(), 1);  vector<int> prev(nums.size(), -1);  int maxc = 0; int idx = -1;  for (int k = 0; k < nums.size(); ++k) {  for (int j = 0; j < k; ++j) {  if (nums[k]%nums[j] == 0) {  if (count[k] < count[j]+1) {  count[k] = count[j]+1;  prev[k] = j;  }  }  }  if (maxc < count[k]) {  idx = k;  maxc = count[k];  }  }    vector<int> result;  while (idx >= 0) {  result.push\_back(nums[idx]);  idx = prev[idx];  }  return result;  } |

# 369. Plus One Linked List [Locked]

|  |
| --- |
| Given a non-negative number represented as a singly linked list of digits, plus one to the number.  The digits are stored such that the most significant digit is at the head of the list. |
| Example:  Input: 1->2->3  Output: 1->2->4 |

**Solution: Linkedlist reverse**

先把链表逆序。然后把头上所有9置零。如果已经到链表尾，则append1.否则当前链表元素+1。最后重新把链表逆序。

|  |
| --- |
| **ListNode\* plusOne(ListNode\* head)** {  ListNode \*l = reverse(head);  ListNode \*p = l;  for (; p->val == 9; p = p->next ) {  p->val = 0;  if (p->next == NULL) p->next = new ListNode(0);  }  p->val++;  return reverse(l);  }  ListNode \*reverse(ListNode\* head) {  ListNode \*tail = NULL;  while (head) {  ListNode \*next = head->next;  head->next = tail;  tail = head;  head = next;  }  return tail;  } |

# 370. Range Addition [Locked]

|  |
| --- |
| Assume you have an array of length n initialized with all 0's and are given k update operations.  Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc.  Return the modified array after all k operations were executed. |
| Example:  Given:  length = 5,  updates = [  [1, 3, 2],  [2, 4, 3],  [0, 2, -2]  ]  Output: [-2, 0, 3, 5, 3] |

**Solution1：segment tree**

用一个segment tree检索区间。并把相应nodes增加inc。最后对每个位置，取它到根结点路径上的node之和。

|  |
| --- |
| inline unsigned int next\_power\_of\_two(unsigned int v) {  --v;  v |= (v >> 1);  v |= (v >> 2);  v |= (v >> 4);  v |= (v >> 8);  v |= (v >> 16);  return v+1;  }  **vector<int> getModifiedArray(int length, vector<vector<int>>& updates)** {  int M = next\_power\_of\_two(length+2);  vector<int> tree(M\*2, 0);  for (vector<int>& v : updates) {  int val = v[2];  for (int i = v[0]+M, j = v[1]+M+2; i ^ j ^ 1; i >>= 1, j >>=1 ) {  if (!(i&1)) tree[i^1] += val;  if (j&1) tree[j^1] += val;  }  }    vector<int> result(length, 0);  for (int i = 1; i <= length; ++i) {  int sum = 0;  for (int j = M+i; j > 0; j >>= 1) sum += tree[j];  result[i-1] = sum;  }  return result;  } |

**Solution2：cumsum**

Solution1 对于静态问题过于复杂。还有一个简单的做法是考虑每次update的first-order derivative是稀疏的。因此只要先在两端update，最后再做cumsum就好。

|  |
| --- |
| **vector<int> getModifiedArray(int length, vector<vector<int>>& updates)** {  vector<int> result(length+1, 0);  for (auto& u : updates) {  result[u[0]] += u[2];  result[u[1]+1] -= u[2];  }  int last = 0;  for (int& v : result) {  last += v;  v = last;  }    result.pop\_back();  return result;  } |

# 371. Sum of Two Integers

|  |
| --- |
| Calculate the sum of two integers a & b, but you are not allowed to use the operator + and -. |
| Example:  Given a = 1 and b = 2, return 3. |

**Solution: recursion**

算法：不考虑进位，a+b得到的是a ^ b，进位是(a & b)<<1。这样问题转化为这两者的和。

递归深度：因为每次加和最后导致进位的一位会向前缩减，而最高位不大于a, b的较高位＋1。因此最多用log2(max(a, b))次操作进位会变为0.

|  |
| --- |
| **int getSum(int a, int b)** {  return b==0? a : getSum(a ^ b, (a & b)<<1);  } |

**Solution: iteration**

|  |
| --- |
| **int getSum(int a, int b)** {  while (b) {  int sum\_without\_carry = (a ^ b);  b = (a & b) << 1; //carry  a = sum\_without\_carry;  }  return a;  } |

# 372. Super Pow

|  |
| --- |
| Your task is to calculate ab mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.  Example1: a = 2 b = [3] Result: 8  Example2: a = 2 b = [1,0] Result: 1024 |

Solution: Math + iteration

问题化简：我们使用两个原则：

* pow(c+r, n) % c = pow(r, n) %c
* ((nr+c1) \*(mr+c2))%c = (c1\*c2)%c

这样pow(a, [b0, b1, ...bN-1]) = pow(a, (b0\*10+b1)\*10 + … +bN-1)\*10 + bN-1)

因此我们可以迭代地求

pow(a, [b0, b1, ...bk]) =pow(a, [b0, b1, …bk-1]\*10+bk]) = pow(pow(a, [b0, b1, …]), 10) \* pow(a, bk)

其中x(k-1) = pow(a, [b0, b1, …bk-1])是上一步迭代的结果。

PowMod：我们提取出该函数完成pow(x, k) % c的运算

x(k)%c = (powmod(x(k-1), 10, c) \* powmod(a, bk, c))%c

powmod (a, b, c) = ab%c是本题的一位数版本，输出范围在[0， c-1]。

在powmod的迭代计算中也用类似方法避免溢出.

|  |
| --- |
| int superPow(int a, vector<int>& b) {  const int c = 1337;  a %= c;  int remainder = 1;  for (int v : b) remainder = (powMod(remainder, 10) \* powMod(a, v)) %c;  return remainder;  }  int powMod(int x, int k) {  const int c = 1337;  int remainder = 1;  for (; k>0; k >>=1, x = (x\*x)%c)  if (k & 1) remainder = (remainder \* x) % c;  return remainder;  } |

# 373. Find K pairs of smallest sum

|  |
| --- |
| You are given two integer arrays nums1 and nums2 sorted in ascending order and an integer k.  Define a pair (u,v) which consists of one element from the first array and one element from the second array.  Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums. |
| Example 1:  Given nums1 = [1,7,11], nums2 = [2,4,6], k = 3  Return: [1,2],[1,4],[1,6]  The first 3 pairs are returned from the sequence:  [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]  Example 2:  Given nums1 = [1,1,2], nums2 = [1,2,3], k = 2  Return: [1,1],[1,1]  The first 2 pairs are returned from the sequence:  [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]  Example 3:  Given nums1 = [1,2], nums2 = [3], k = 3  Return: [1,3],[2,3]  All possible pairs are returned from the sequence:  [1,3],[2,3] |

**Solution: k-way merge sort**

我们考虑一个由nums1所有元素和nums2所有元素之和形成的表，例如：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 1 | 2 |
| 1 | 2 | 2 | 3 |
| 2 | 3 | 3 | 4 |

在此表中，每行都是顺序排列的。如果我们想求它的顺序排列，可以merge sort所有行。

|  |
| --- |
| **vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k)** {  if (nums1.size() == 0 || nums2.size() == 0) return vector<pair<int, int>>{};  auto cmp = [&nums1, &nums2](pair<int, int>p1, pair<int, int>p2) {  return nums1[p1.first]+nums2[p1.second] > nums1[p2.first]+nums2[p2.second];  };  priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp)> heap(cmp);    for (int j = 0; j < nums1.size(); ++j) heap.emplace(j, 0);    vector<pair<int, int>> result;  for (int j = 0; j < k && !heap.empty(); ++j) {  int idx1 = heap.top().first, idx2 = heap.top().second;  heap.pop();  result.emplace\_back(nums1[idx1], nums2[idx2]);  if (idx2 < nums2.size()-1) heap.emplace(idx1, idx2+1);  }  return result;  } |

# 374. Guess Number Higher or Lower

|  |
| --- |
| We are playing the Guess Game. The game is as follows:  I pick a number from 1 to n. You have to guess which number I picked.  Every time you guess wrong, I'll tell you whether the number is higher or lower.  You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):  -1 : My number is lower  1 : My number is higher  0 : Congrats! You got it! |
| Example: n = 10, I pick 6. Return 6. |

**Solution: binary search**

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

# 375. Guess Number Higher or Lower II

|  |
| --- |
| We are playing the Guess Game. The game is as follows:  I pick a number from 1 to n. You have to guess which number I picked.  Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.  However, when you guess a particular number x, and you guess wrong, you pay $x. You win the game when you guess the number I picked.  Given a particular n ≥ 1, find out how much money you need to have to guarantee a win. |
| Example:  n = 10, I pick 8.  First round: You guess 5, I tell you that it's higher. You pay $5.  Second round: You guess 7, I tell you that it's higher. You pay $7.  Third round: You guess 9, I tell you that it's lower. You pay $9.  Game over. 8 is the number I picked.  You end up paying $5 + $7 + $9 = $21. |

**Solution：memorized recursion**

minmax:

对猜数字的人来说，猜测可以有不同的策略（比如顺序猜、二分等等），这个策略跟target是几没有关系。但对每个策略，根据target不同可能产生不同的代价。

minmax里的max是指给定一个策略，需要考虑最坏情况。例如假设策略是顺序猜，应该考虑要猜的数是n的可能性。min是指在所有可能的策略中，取最坏情况最小的一种作为最优策略。

所以你要求的是min\_{所有策略} （max\_{所有target} 这个策略和这个target所产生的代价）

memorized recursion:

把要猜的数的范围从1-n推广到p-q。那么对每个pair (p, q)，都有一个minmax cost，假设这个是一个函数C(p, q)。这就是我们要计算的dp数组。

很显然地如果p>=q，也就是范围里只有0-1个数，则不产生代价。

当p<q时，假设第一次猜的是x，则可以根据target所在范围分三种情况，按照问题定义各自产生不同的代价。这里我们还是求minmax：给定x，我们需要考虑所有target的最坏情况，在最外层需要考虑所有x的取值，选其中代价最小的一个（最优策略）。

|  |
| --- |
| int getMoneyAmount(int n) {  vector<vector<int>> amount(n+1, vector<int>(n+1, 0));  return getMoneyAmount\_recur(1, n, amount);  }  int getMoneyAmount\_recur(int l, int r, vector<vector<int>> &amount) {  if (l >= r || r <= 0 || l >= amount.size()) return 0;  if (amount[l][r] > 0) return amount[l][r];    amount[l][r] = INT\_MAX;  for (int j = l; j <=r; ++j)  amount[l][r] = min(amount[l][r],  j + max(getMoneyAmount\_recur(l, j-1, amount),  getMoneyAmount\_recur(j+1, r, amount)));  return amount[l][r];  } |

# 376. Wiggle subsequence

|  |
| --- |
| A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.  For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.  Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.  Follow up: Can you do it in O(n) time? |
| Examples:  Input: [1,7,4,9,2,5]  Output: 6  The entire sequence is a wiggle sequence.  Input: [1,17,5,10,13,15,10,5,16,8]  Output: 7  There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8].  Input: [1,2,3,4,5,6,7,8,9]  Output: 2 |

**Solution：**

单调区间至多只能取两个点。如要使之后的wiggle sequence尽量长，则尽量取单调区间的两端。

因此最长wiggle subsequence为sequence的所有拐点，再加上左右两端各随便取一个点。

|  |
| --- |
| int wiggleMaxLength(vector<int>& nums) {  if (nums.size() <= 1) return nums.size();    int count = 1, p = 0;  while (p < nums.size() && nums[p] == nums[0]) ++p;    for (; p < nums.size(); ++count) {  int diff = nums[p] - nums[p-1];  int q = p;  while ( q < nums.size() &&  ( (diff ^(nums[q]-nums[q-1])) >= 0 || nums[q] == nums[q-1])  )  ++q;  p = q;  }  return count;  } |

# 

# 377. Combination Sum IV

|  |
| --- |
| Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.  Example:  nums = [1, 2, 3] target = 4  The possible combination ways are:  (1, 1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 3), (2, 1, 1), (2, 2), (3, 1)  Note that different sequences are counted as different combinations.  Therefore the output is 7.  Follow up:  What if negative numbers are allowed in the given array?  How does it change the problem?  What limitation we need to add to the question to allow negative numbers? |

**Solution 1: recursion**

题意分析：需注意从例子看(1, 1, 2) != (2, 1, 1)，所以要考虑每种分法的所有permutation。

递归结构：穷举加法中的第一个数v，递归调用计算构成target-v的所有可能性(要求v<=target)

结束条件：target = 0，返回1

边界条件：target <= 0，直接返回0，注意这和递归结束条件不同

|  |
| --- |
| int combinationSum4(vector<int>& nums, int target) {  if (target<=0) return 0;  return combinationSumRecur(nums, target);  }  int combinationSumRecur(vector<int>& nums, int target) {  if (target == 0) return 1;  int count = 0;  for (int v : nums)  if (v <= target) count += combinationSumRecur(nums, target-v);  return count;  } |

**Solution 2: memorized recursion**

把combinationSumRecur(nums, t)的所有结果cache下来供以后调用。

|  |
| --- |
| int combinationSum4(vector<int>& nums, int target) {  if (target<=0) return 0;  vector<int> cache(target+1, -1);  cache[0] = 1;  return combinationSumRecur(nums, target, cache);  }  int combinationSumRecur(vector<int>& nums, int target, vector<int>& cache) {  if (cache[target]>=0) return cache[target];  int count = 0;  for (int v : nums) if (v <= target) count += combinationSumRecur(nums, target-v)  return (cache[target] = count);  } |

**Solution 3: DP**

特别当nums包含1时，所有t值都会被访问到。因此不如直接动态规划。

|  |
| --- |
| int combinationSum4(vector<int>& nums, int target) {  if (target<=0) return 0;  vector<int> count(target+1, 0);  count[0] = 1;  for (int t = 1; t <= target; ++t) {  for (int v : nums) {  if (v <=t) count[t] += count[t - v];  }  }  return count.back();  } |

**Follow up:**

如果存在一个nums的子集和为0,则这个子集合可以出现任意多次。要使得返回值不为无穷大，可以要求每个加法项最多包括一个和为0的子集。

# 378

# 379

# 

# 380

# 

# 381

# 

# 382

# 

# 383

# 

# 384

# 

# 385

# 

# 386

# 

# 387

# 

# 388

# 

# 389

# 

# 390

# 

# 391. Perfect Rectangle

|  |
| --- |
| Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.  Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)). |
| Example 1: rectangles = [ [1,1,3,3], [3,1,4,2], [3,2,4,4], [1,3,2,4], [2,3,3,4]]  Return true. All 5 rectangles together form an exact cover of a rectangular region.  Example 2: rectangles = [ [1,1,2,3], [1,3,2,4], [3,1,4,2], [3,2,4,4]]  Return false. Because there is a gap between the two rectangular regions.  Example 3: rectangles = [ [1,1,3,3], [3,1,4,2], [1,3,2,4], [3,2,4,4]]  Return false. Because there is a gap in the top center.  Example 4: rectangles = [ [1,1,3,3], [3,1,4,2], [1,3,2,4], [2,2,4,4]]  Return false. Because two of the rectangles overlap with each other. |

**Sol1: brute-force**

两两检查矩形是否overlap，如果overlap，就返回false。

在扫矩形的同时计算所有矩形的面积总和和bounding box，最后返回面积总和是否等于bounding box.

**Sol2: line sweeping**

把x轴当作时间。y轴当作资源。矩形[l, r, t, b]认为是要求时间l把资源[t, b]分配出去。时间r再收回。

这样检查overlap的意思就是看当某个矩形要获得资源[t, b]时，会不会其中有一段已经被分配掉了还没回收，如图。这只需要maintain一个有序的interval结构就可以了（用set)。

有一个有点小麻烦的地方是在x位置可能同时有右边缘和左边缘。这时，需要确保右边缘把interval从set里删掉，再加入左边缘。可以使用的trick是把位置在x的右边缘放在时间t=2x，左边缘放在t=2x+1。这样所有左右边缘的顺序不变，但右边缘一定早于左边缘被处理。

**A**

**B**

**A**

**B**

**C**

**C**

x=2

x=3

t=5

t=6

t=7

例如右图：

如果不作mapping，访问顺序可能是

A(start), C(start), A(end), B(start),...

也可能是

A(start), C(start), B(start), A(end) -> 产生冲突

如果作mapping，访问顺序一定是

A(start), C(start), A(end), B(start)

|  |
| --- |
| struct interval {  int start;  int end;  interval(int start\_, int end\_) : start(start\_), end(end\_) {};  };  struct edge {  int t;  interval i;  edge(int t\_, interval i\_) : t(t\_), i(i\_) {};  };  struct interval\_cmp {  bool operator()(interval i1, interval i2) { return i1.start < i2.start; };  };  struct edge\_cmp {  bool operator()(edge e1, edge e2) { return e1.t > e2.t; };  };  **bool isRectangleCover(vector<vector<int>>& rectangles)** {  priority\_queue<edge, vector<edge>, edge\_cmp> q;  set<interval, interval\_cmp> active\_intervals;    int minx = INT\_MAX, miny = INT\_MAX, maxx = INT\_MIN, maxy = INT\_MIN;  int area = 0;  for (const auto& rect : rectangles) {  area += (rect[2]-rect[0])\* (rect[3]-rect[1]);  minx = min(rect[0], minx);  miny = min(rect[1], miny);  maxx = max(rect[2], maxx);  maxy = max(rect[3], maxy);  q.emplace(rect[0]\*2+1, interval(rect[1], rect[3]));  q.emplace(rect[2]\*2 , interval(rect[1], rect[3]));  }    while (!q.empty()) {  int t = q.top().t;  interval i = q.top().i;  if (t % 2) { //insert interval  auto it = active\_intervals.lower\_bound(i);  if (it != active\_intervals.begin() && prev(it)->end > i.start) return false;  if (it != active\_intervals.end() && it->start < i.end) return false;  active\_intervals.insert(it, i);  }  else { //remove interval  active\_intervals.erase(i);  }  q.pop();  }  return area == (maxx-minx) \* (maxy - miny);  } |

**Sol3: corner pattern**

每个顶点可能被claim为一个矩形的左上角、左下角、右上角、右下角。

如果它的某个象限同时被多个矩形占据，则说明有overlap，一定返回false.

否则顶点可以根据每个象限有没有被claim分为16种情况(4个bits，每个bit表示顶点的一个象限)。如果我们把旋转对称的情况合并到一起可以细分成5种结构，其中有3种可能出现在perfect cover里。如下图的C, T, X三种pattern。

C

T

X

L

D

因此我们只要验证所有顶点

(1) 在bounding box corner上 (此时可以保证是C pattern)

(2) 是T pattern（4-bit表示为3, 5, 10, 12）或者X pattern (4-bit表示为15)

|  |
| --- |
| struct pairhash {//double hash function for pair key  public:  template <typename T, typename U>  size\_t operator()(const pair<T, U> &rhs) const {  size\_t l = hash<T>()(rhs.first);  size\_t r = hash<U>()(rhs.second);  return l + 0x9e3779b9 + (r << 6) + (r >> 2);  }  };  **bool isRectangleCover(vector<vector<int>>& rectangles)** {  // step 1: counting  unordered\_map<pair<int, int>, int, pairhash> corner\_count;  int minx = INT\_MAX, maxx=INT\_MIN, miny=INT\_MAX, maxy=INT\_MIN;  for (auto& rect : rectangles) {  minx = min(minx, rect[0]);  maxx = max(maxx, rect[2]);  miny = min(miny, rect[1]);  maxy = max(maxy, rect[3]);    int& m1 = corner\_count[make\_pair(rect[0], rect[1])];  if (m1 & 1) return false; else m1 |= 1;  int& m2 = corner\_count[make\_pair(rect[2], rect[1])];  if (m2 & 2) return false; else m2 |= 2;  int& m3 = corner\_count[make\_pair(rect[0], rect[3])];  if (m3 & 4) return false; else m3 |= 4;  int& m4 = corner\_count[make\_pair(rect[2], rect[3])];  if (m4 & 8) return false; else m4 |= 8;  }    //step2: checking  for (const auto& kv: corner\_count) {  pair<int, int> pos; int mask;  tie(pos, mask) = kv;  if ((pos.first != minx && pos.first != maxx) ||  (pos.second != miny && pos.second != maxy)) //not corner of bounding box  if (mask != 3 && mask != 5 && mask != 10 && mask != 12 && mask != 15)  return false;  }  return true;  } |