求和问题总结(leetcode 2Sum, 3Sum, 4Sum, K Sum)

发表于 2013 年 1 月 8 日 作者 sigmainfy 浏览次数：1,612 次 — 12 条评论 ↓

前言:

做过leetcode的人都知道, 里面有2sum, 3sum(closest), 4sum等问题, 这些也是面试里面经典的问题, 考察是否能够合理利用排序这个性质, 一步一步得到高效的算法. 经过总结, 本人觉得这些问题都可以使用一个通用的K sum求和问题加以概括消化, 这里我们先直接给出K Sum的问题描述和算法(递归解法), 然后将这个一般性的方法套用到具体的K, 比如leetcode中的2Sum, 3Sum, 4Sum问题. 同时我们也给出另一种哈希算法的讨论.

leetcode求和问题描述(K sum problem):

K sum的求和问题一般是这样子描述的：给你一组N个数字(比如 vector<int> num), 然后给你一个常数(比如 int target) ，我们的goal是在这一堆数里面找到K个数字，使得这K个数字的和等于target。

注意事项(constraints):

注意这一组数字可能有重复项：比如 1 1 2 3 , 求3sum, 然后 target = 6, 你搜的时候可能会得到两组1 2 3, 1 2 3，1 来自第一个1或者第二个1, 但是结果其实只有一组，所以最后结果要去重。

K Sum求解方法, 适用leetcode 2Sum, 3Sum, 4Sum:

方法一： 暴力，就是枚举所有的K-subset, 那么这样的复杂度就是 从N选出K个，复杂度是O(N^K)

方法二： 排序，这个算法可以考虑最简单的case, 2sum，这是个经典问题，方法就是先排序，然后利用头尾指针找到两个数使得他们的和等于target, 这个2sum算法网上一搜就有，这里不赘述了，给出2sum的核心代码：

```
01 //2 sum
02 int i = starting; //头指针
03 int j = num.size() - 1; //尾指针
04 while(i < j) {
05     int sum = num[i] + num[j];
06     if(sum == target) {
07         store num[i] and num[j] somewhere;
08         if(we need only one such pair of numbers)
09             break;
10         otherwise
11             do ++i, --j;
12     }
13     else if(sum < target)
14         ++i;
15     else
16         --j;
17 }
```

2sum的算法复杂度是O(N log N) 因为排序用了N log N以及头尾指针的搜索是线性的，所以总体

搜索... 搜

本站公告

1. 除非另有声明，本站内容都遵循采用CC BY-NC-SA 协议发布。文章版权归原作者所有, 转载请注明来自烟客旅人 sigmainfy 。

2. 若直接点击本站RSS链接出错，请右键点击 ”文章RSS”或”评论RSS”连接，选择”复制连接地址 / copy link address”，然后打开你的阅读器，黏贴该链接地址到阅读器即可完成订阅。

分类目录

- Android
- C/C++
- Java
- Javascript
- leetcode
- Linux
- PAT
- Python
- WindowsProgramming
- WordPress
- 其他
- 技术
- 数据结构和算法
- 癫笑痴狂
- 编程语言
- 面试题选编

标签云

Akismet bfs C/C++ chrome dfs domain name dp GA Godaddy greedy hash ip被封 Java kick off like linear linked list linux n/a pagerank permalink plugin PostViews pr referral code RSS

是O(N log N)，好了现在考虑3sum, 有了2sum其实3sum就不难了，这样想：先取出一个数，那么我只要在剩下的数字里面找到两个数字使得他们的和等于(target – 那个取出的数)就可以了吧。所以3sum就退化成了2sum, 取出一个数字，这样的数字有N个，所以3sum的算法复杂度就是O(N^2), 注意这里复杂度是N平方，因为你排序只需要排一次，后面的工作都是取出一个数字，然后找剩下的两个数字，找两个数字是2sum用头尾指针线性扫，这里很容易错误的将复杂度算成O(N^2 log N)，这个是不对的。我们继续的话4sum也就可以退化成3sum问题，那么以此类推，K-sum一步一步退化，最后也就是解决一个2sum的问题，K sum的复杂度是O(n^(K-1))。这个界好像是最好的界了，也就是K-sum问题最好也就能做到O(n^(K-1))复杂度，之前有看到过有人说可以严格数学证明，这里就不深入研究了。

更新: 感谢网友Hatch提供他的K Sum源代码, 经供参考:

```
01  class Solution {
02  public:
03      vector< vector > findZeroSumInSortedArr(vector &num, int
    begin, int count, int target)
04      {
05          vector<vector > ret;
06          vector tuple;
07          set visited;
08          if (count == 2)
09          {
10              int i = begin, j = num.size()-1;
11              while (i < j)
12              {
13                  int sum = num[i] + num[j];
14                  if (sum == target && visited.find(num[i]) ==
    visited.end())
15                  {
16                      tuple.clear();
17                      visited.insert(num[i]);
18                      visited.insert(num[j]);
19                      tuple.push_back(num[i]);
20                      tuple.push_back(num[j]);
21                      ret.push_back(tuple);
22                      i++; j-;
23                  }
24                  else if (sum < target)
25                  {
26                      i++;
27                  }
28                  else
29                  {
30                      j-;
31                  }
32              }
33          }
34          else
35          {
36              for (int i=begin; i<num.size(); i++)
37              {
38                  if (visited.find(num[i]) == visited.end())
39                  {
40                      visited.insert(num[i]);
41                      vector subRet = findZeroSumInSortedArr(num,
    i+1, count-1, target-num[i]);
42                      if (!subRet.empty())
43                      {
44                          for (int j=0; j<subRet.size(); j++)
45                          {
46                              subRet[j].insert(subRet[j].begin(),
    num[i]);
47                          }
48
49                          ret.insert(ret.end(), subRet.begin(),
    subRet.end());
50                      }
51                  }
52              }
53          }
54
55          return ret;
56      }
57
58      vector threeSum(vector &num) {
59          sort(num.begin(), num.end());
60          return findZeroSumInSortedArr(num, 0, 3, 0);
61      }
62
63      vector fourSum(vector &num, int target) {
64          sort(num.begin(), num.end());
65          return findZeroSumInSortedArr(num, 0, 4, target);
66      }
67  };
```

K Sum (2Sum, 3Sum, 4Sum) 算法优化 (Optimization)：

这里讲两点，第一，注意比如3sum的时候，先整体排一次序，然后枚举第三个数字的时候不需要重复， 比如排好序以后的数字是 a b c d e f, 那么第一次枚举a, 在剩下的b c d e f中进行2 sum, 完了以后第二次枚举b, 只需要在 c d e f中进行2sum好了，而不是在a c d e f中进行2sum, 这个大家可以自己体会一下，想通了还是挺有帮助的。第二，K Sum可以写一个递归程序很优雅的解决，具体大家可以自己试一试。写递归的时候注意不要重复排序就行了。

K Sum (2Sum, 3Sum, 4Sum) 算法之3sum源代码(不使用std::set)和相关开放问题讨论：

因为已经收到好几个网友的邮件需要3sum的源代码, 那么还是贴一下吧, 下面的代码是可以通过leetcode OJ的代码(又重新写了一遍, 于Jan, 11, 2014 Accepted), 就当是K sum的完整的一个case study吧, 顺便解释一下上面的排序这个注意点, 同时我也有关于结果去重的问题可以和大家讨论一下, 也请大家集思广益, 发表意见, 首先看源代码如下：

```
01 class Solution {
02 public:
03     vector threeSum(vector &num) {
04         vector vecResult;
05         if(num.size() < 3)
06             return vecResult;
07 
08         vector vecTriple(3, 0);
09         sort(num.begin(), num.end());
10         int iCurrentValue = num[0];
11         int iCount = num.size() - 2; // (1) trick 1
12         for(int i = 0; i < iCount; ++i) {
13             if(i && num[i] == iCurrentValue) { // (2) trick 2: trying to avoid repeating triples
14                 continue;
15             }
16             // do 2 sum
17             vecTriple[0] = num[i];
18             int j = i + 1;
19             int k = num.size() - 1;
20             while(j < k) {
21                 int iSum = num[j] + num[k];
22                 if(iSum + vecTriple[0] == 0) {
23                     vecTriple[1] = num[j];
24                     vecTriple[2] = num[k];
25                     vecResult.push_back(vecTriple); // copy constructor
26                     ++j;
27                     --k;
28                 }
29                 else if(iSum + vecTriple[0] < 0)
30                     ++j;
31                 else
32                     --k;
33             }
34             iCurrentValue = num[i];
35         }
36                 // trick 3: indeed remove all repeated triplets
37                 // trick 4: already sorted, no need to sort the triplets at all, think about why?
38         vector< vector >::iterator it = unique(vecResult.begin(), vecResult.end());
39         vecResult.resize( distance(vecResult.begin(), it) );
40         return vecResult;
41     }
42 };
```

首先呢, 在K Sum问题中都有个结果去重的问题, 前文也说了, 如果输入中就有重复元素的话, 最后结果都需要去重, 去重有好几个办法, 可以利用std::set的性质(如leetcode上3sum的文章, 但是他那个文章的问题是, set没用好, 导致最终复杂度其实是O(N^2 * log N), 而非真正的O(N^2)), 可以利用排序(如本文的方法)等, 去重本身不难, 难的是不利用任何排序或者std::set直接得到没有重复的triplet结果集. 本人试图通过已经排好序这个性质来做到这一点(试图不用trick 3和4下面的两条语句)， 但是经过验证这样的代码(没有trick 3, 4下面的两行代码, 直接return vecResult)也不能保证结果没有重复，于是不得不加上了trick 3, 4，还是需要通过在结果集上进一步去重. **笔者对于这个问题一直没有很好的想法, 希望这里的代码能抛砖引玉, 大家也讨论一下有没有办法, 或者利用排序的性质或者利用其它方法, 直接得到没有重复元素的triplet结果集, 不需要去重这个步骤.**

那么还是解释一下源代码里面有四个trick, **以及笔者试图不利用任何std::set或者排序而做到去重的想法**. 第一个无关紧要顺带的小trick 1, 是说我们排好序以后, 只需要检测到倒数第三个数字就行了, 因为剩下的只有一种triplet 由最后三个数字组成. 接下来三个trick都是和排序以及最后结果的去重问题有关的, 我一起说.

笔者为了达到不需要在最后的结果集做额外的去重, 尝试了以下努力: 首先对输入数组整体排序, 然后使用之前提到的3sum的算法, 每次按照顺序先定下triplet的第一个数字, 然后在数组后面寻

和UTC的关系以及如何相互转换(二)》
- sigmainfy 发表在《PAT 解题报告 1074. Reversing Linked List (25)》
- leetcode 发表在《求和问题总结(leetcode 2Sum, 3Sum, 4Sum, K Sum)》
- sigmainfy 发表在《求和问题总结(leetcode 2Sum, 3Sum, 4Sum, K Sum)》
- sigmainfy 发表在《求和问题总结(leetcode 2Sum, 3Sum, 4Sum, K Sum)》
- Hatch 发表在《求和问题总结(leetcode 2Sum, 3Sum, 4Sum, K Sum)》
- AmazingCaddy 发表在《求和问题总结(leetcode 2Sum, 3Sum, 4Sum, K Sum)》

文章归档

- 2014 年三月
- 2014 年二月
- 2014 年一月
- 2013 年十二月
- 2013 年十一月
- 2013 年十月
- 2013 年九月
- 2013 年八月
- 2013 年七月
- 2013 年六月
- 2013 年五月
- 2013 年四月
- 2013 年三月
- 2013 年二月
- 2013 年一月
- 2012 年十二月

功能

- 登录
- 文章 RSS
- 评论 RSS
- WordPress.org

找2sum, 由于已经排好序, 为了防止重复, 我们要保证triplet的第一个数字没有重复, 举个例子, -3, – 3, 2, 1, 那么第二个-3不应该再被选为我们的第一个数字了, **因为在第一个-3定下来寻找2sum的时候, 我们一定已经找到了所有以-3为第一个数字的triplet(trick 2).** 但是这样做尽管可以避免一部分的重复, 但是还有另一种重复无法避免: -3, -3, -3, 6, 那么在定下第一个-3的时候, 我们已经有两组重复triplet <-3, -3, 6>， 如何在不使用std::set的情况下避免这类重复, 笔者至今没有很好的想法. **大家有和建议？望不吝赐教！**

更新: 感谢网友stayshan的留言提醒, 根据他的留言, 不用在最后再去重. 于是可以把trick 3, 4下面的两行代码去掉, 然后把while里面的copy constructor这条语句加上是否和前一个元素重复的判断变成下面的代码就行了.

这样的做法当然比我上面的代码更加优雅, 虽然本质其实是一样的, 只不过去重的阶段变化了, 进一步的, 我想探讨的是, 我们能不能通过”不产生任何重复的triplet”的方法直接得到没有重复的triplet集合? 网友stayshan提到的方法其实还是可能生成重复的triplet, 然后通过和已有的triplet集合判断去重, **笔者在这里试图所做的尝试更加确切的讲是想找到一种方法, 可以保证不生成重复的triplet. 现有的方法似乎都是post-processing, i.e., 生成了重复的triplet以后进行去重. 笔者想在这里探讨从而找到一种我觉得可以叫他pre-processing的方法, 能够通过一定的性质(可能是排序的性质等)保证不会生成triplet, 从而达到不需任何去重的后处理(post-processing)手段. 感觉笔者抛出的砖已经引出了挺好的思路了啊, 太好了, 大家有啥更好的建议, 还请指教啊 ： ）**

```
01 class Solution {
02 public:
03     vector threeSum(vector &num) {
04         // same as above
05                 // ...
06         for(int i = 0; i < iCount; ++i) {
07             // same as above
08                         // ...
09             while(j < k) {
10                 int iSum = num[j] + num[k];
11                 if(iSum + vecTriple[0] == 0) {
12                     vecTriple[1] = num[j];
13                     vecTriple[2] = num[k];
14                     if(vecResult.size() == 0 || vecTriple !=
   vecResult[vecResult.size() - 1])
15                         vecResult.push_back(vecTriple); // copy
   constructor
16                     ++j;
17                     --k;
18                 }
19                 else if(iSum + vecTriple[0] < 0)
20                     ++j;
21                 else
22                     --k;
23             }
24             iCurrentValue = num[i];
25         }
26         return vecResult;
27     }
28 };
```

Hash解法(Other)：

其实比如2sum还是有线性解法的，就是用hashmap, 这样你check某个值存在不存在就是常数时间，那么给定一个sum, 只要线性扫描, 对每一个number判断sum – num存在不存在就可以了。注意这个算法对有重复元素的序列也是适用的。比如 2 3 3 4 那么hashtable可以使 hash(2) = 1; hash(3) = 1, hash(4) =1其他都是0，那么check的时候，扫到两次3都是check sum – 3在不在hashtable中，注意最后返回所有符合的pair的时候也还是要去重。这样子推广的话 3sum 其实也有O(N^2)的类似hash算法，这点和之前是没有提高的，但是4sum就会有更快的一个算法。

4sum的hash算法：

O(N^2)把所有pair存入hash表，并且每个hash值下面可以跟一个list做成map， map[hashvalue] = list，每个list中的元素就是一个pair, 这个pair的和就是这个hash值，那么接下来求4sum就变成了在所有的pair value中求 2sum，这个就成了线性算法了，注意这里的线性又是针对pair数量(N^2)的线性，所以整体上这个算法是O(N^2)，而且因为我们挂了list, 所以只要符合4sum的我们都可以找到对应的是哪四个数字。

到这里为止有人提出这个算法不对 (感谢Jun提出这点!! See the comment below), 因为这里的算法似乎无法检查取出来的四个数字是否有重复的, 也就是说在转换成2sum问题得到的那些个pair中, 有可能会有重复元素, 比如说原来数组中的第一个元素其实是重复了两次才使得4 sum满足要求, 那么这样得到的四元组(四个数字的和等于给定的值), 其实只有三个原数组元素, 其中第一个元素用了两次, 那么这样就不对了. 如果仅从我上面的描述来看, 确实是没有办法检查重复的, 但

是仔细想想我们只要在map中存pair的的时候记录下的不是原数组对应的值, 而是原数组的id,就可以避免这个问题了. 更加具体的, map[hashvalue] = list, 每个list的元素就是一个pair, 这个pair<int, int> 中的pair是原来的array id, 使得这两个id对应到元素组中的元素值的和就是这个hash值. 那么问题就没有了, 我们在转换成的2sum寻找所有pair value的2sum的同时要检查最后得到的四元组<id1, id2, id3, id4>没有重复id. 这样问题就解决了.

结束语：

这篇文章主要想从一般的K sum问题的角度总结那些比较经典的求和问题比如leetcode里面的2sum, 3sum(closest), 4sum等问题，文章先直接给出K Sum的问题描述和算法(递归解法), 然后将这个一般性的方法套用到具体的K, 比如leetcode中的2Sum, 3Sum, 4Sum问题. 同时我们也给出另一种哈希算法的讨论. 那么这篇文章基本上还是自己想到什么写什么，有疏忽不对的地方请大家指正，也欢迎留言讨论，如果需要源代码，请留言或者发邮件到info@tech-wonderland.net

（全文完，原创文章，转载时请注明作者和出处）

0

Related Posts

- PAT 解题报告 1007. Maximum Subsequence Sum (25)
- leetcode: Search in Rotated Sorted Array II
- PAT 解题报告 1032. Sharing (25)
- leetcode: Count and Say
- PAT 解题报告 1049. Counting Ones (30)