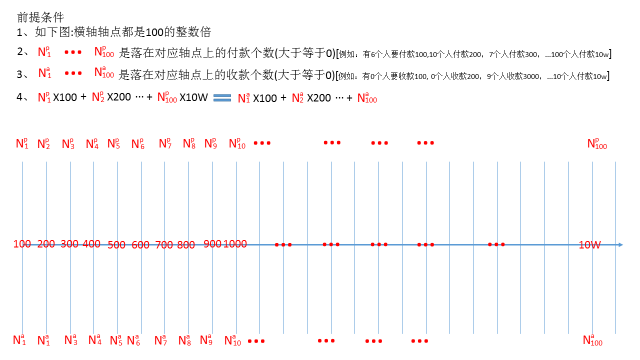
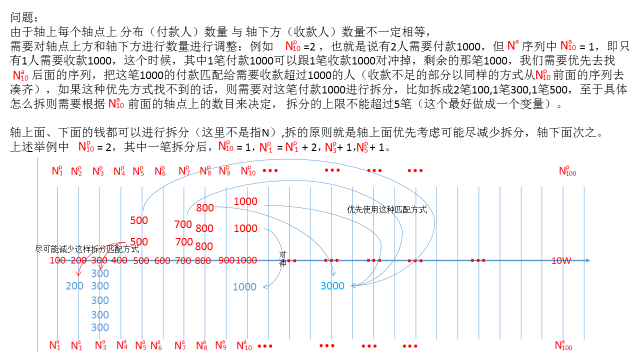
**匹配需求算法设计**

# 问题分析

问题描述如图1和图2。假设收款人构成集合A，打款人构成集合B。



**图1 问题前提条件**



**图2 问题说明**

将集合A，集合B中元素按照额度从小到大排序，将集合A和集合B中相等额度的元素直接抵消，剩下的集合中不再有相等的元素。

从两个集合中最大的数分析，假设集合A中最大的数大于集合B中最大的数，那么将被拆分（在本问题下，即为至少要收款两次）。在此情况下，被拆分存在被2拆分，3拆分，4拆分及以上等情况，其中3拆分此可看成是由2次的2拆分，过程如下：

**A->E,F,G == A->E,C , C->F,G**

对一个数拆分的问题可归结为子集和问题（子集和问题：在一个集合找到子集，使其满足子集所有元素和为给定的值），子集和问题是一个NP完全问题，目前没有任何一个够快的方法可以在合理的时间内（意即多项式时间）找到答案。只能一个个将它的子集取出来一一测试，它的时间复杂度是，N是此集合的元素数量。

对于我们这个问题，每一个元素都有可能被拆分，考虑所有元素的拆分情况，是一个比子集和更复杂的问题，即对每一个数的操作都是一个子集和的问题，要找到最优解只有穷举所有的可能，目前是无法实现的。对于此问题，我们的目标是找一个次优解，且其时间复杂度不能太大。

本文中所述的一次操作或一次交易均表示一次打款操作或是一个收款操作。

# 算法设计

## 2.1算法思路说明

基于此问题，我们依据：局部最优兼顾全局；设计的算法思路如下：

|  |
| --- |
| **Step1：对集合A和集合B由小到大排序**  **Step2：对冲抵消集合A和集合B中额度相等的元素，更新集合A集合B**  **Step3-1：若>则：从集合B中寻找一个2元素子集，其元素额度和为；若能找到，则将拆分为最先找到的子集的两个元素，更新集合A，集合B；若不能找到，则先将抵消，元素对应的额度减小并依据其大小插入到集合A中，更新集合B**  **Step3-2: 若>则：从集合A中寻找一个2元素子集，其元素额度和为；若能找到，则将拆分为最先找到的子集的两个元素，更新集合A，集合B；若不能找到，则先将抵消，元素对应的额度减小并依据其大小插入到集合B中，更新集合A；**  **Step4:重复step3，直到每个元素都完成匹配**  **Step5:输出匹配结果** |

下面解释为何上述算法可以得到较优的结果。算法整体思想是贪心的思想，贪心算法保证每一步都是最优的即局部最优，因此算法保证每一步的拆分都是最少的。

集合A中元素和集合B中元素地位是对等的，打款的次数必定等于收款的次数，从打款者的视角看是决定打款给那些人最合适，从收款这视角看是从哪些人这收款最合适。

在step3-1中，若>，那么是一定需要被拆分的，被2拆分可以保证其被拆分的次数最少，因此首先判断其是否能被2拆分，如果可以，那么进行2拆分；如果不能被2拆分，按照贪心的思想，应该寻找3拆分，但此处不这么处理，前面提到3拆分可以由两次2拆分构成，因此此处先将其拆分为和，按照顺序插回集合A中，后续操作中集合A中的存在和另一个数一起加和抵消集合B中一个数的可能，若当集合成为集合A中最大的数时，若次数回到Step3-1或Step3-2均存在被一次抵消的可能，若不能一直被抵消（集合B中有数比大时，一直存在被抵消的可能），此时再进行不可避免的2拆分，如此循环。

从以上可以看出整个过程中，对于当前最优的操作则直接执行，对于当前的次优结果则等待执行，直到被抵消，或是由于数据的特殊需要再次拆分。上述算法流程保证了局部最优的同时兼顾整体，达到一个较优的结果。

在几个小数据集的测试中，上述算法基本达到全局最优的结果。

## 2.2算法实现与分析

整个算法用C++实现，下面代码均为C++代码。

### 2.2.1 对象描述

用一个Exchange\_Item类来表示打款对象或收款对象，类的定义如下：

|  |
| --- |
| class Exchange\_Item  {  Public:  …  private:  std::string User\_ID;  int money;// 总收款/打款额度  int initOrder; //初始赋值的顺序，为了最后排序输出和原先顺序一样的结果  int res\_money;// 剩余收款/打款额度  int weight;//权重，每被拆分（交易）一次，权重+1  int max\_weight;//最大权重，也即最大交易次数  std::vector<std::string> exchange\_ID;//与之发生交易的ID的集合  std::vector<int> exchange\_money; //与之发生交易的额度的集合，与ID对应  }; |

Rers为list（链表）容器，收款对象的集合；Ters为list（链表）容器，为打款对象的集合；RersSaver为余额为0的收款对象的集合，TersSaver为余额为0的打款对象的集合。

### 2.2.2 对冲抵消算法设计

由于此问题的需要处理的数据比较大，因此对算法时间复杂度有较高的要求，设计了一个时间复杂度为，其中包括排序操作。

算法思想：首先将两组数据递增排序，用两个迭代器Rp,Tp分别指向Rers和Ters的首个元素，如果Rp指向的元素的余额大于Tp所指向的那么Tp++指向后一位，如果Rp指向的元素的余额小于Tp所指向的那么Rp++指向后一位，找到相等发生交易并记录，继续寻找下一对余额相等的元素，直到Tp指向Ters最后一个元素的下一位或Rp指向指向Rers最后一个元素的下一位，整个过程停止。排序的算法复杂度为，上述查找过程只有一次遍历，算法时间复杂度为。

具体算法实现见dealRepetition(ExContainer& Rers, ExContainer& Ters)

### 2.2.3 存储与容器选择

算法中存在较多的随机访问和几次遍历操作以及大量的中间插入操作，在C++中vector支持快速随机访问但插入操作效率低，list容器不支持随机访问但是插入效率高。测试发现使用list容器效率更高。

由于list容器不支持随机访问元素，因此将处理完的元素（余额为0）的和未处理完的分开存储，处理完的元素容器只需新增元素无需其他操作，使得待处理的list长度随着处理过程逐渐变短，插入操作以及访问更快。

实际测试显示，使用单独一个容器存储，前半部分存储已经处理完的数据，后半部分处理未处理完的数据，使用一个迭代器指示已经处理完成的最后一个元素的下一个来区分处理完成的数据和未处理完成的数据，无论是使用vector容器还是list容器效率均较低。

### 2.2.4 寻找2sum与拆分抵消

对于已经不存在可以对冲抵消的元素的集合A和集合B中，对于一个必须被拆分的元素，2拆分是一个局部最优的结果，2拆分的问题可归结为从一个集合中找出一个二元素子集，使其两元素的和为一个给定的值。

对于本问题，由于集合中元素已经是排好序的，因此可采用如下算法（以数组为例阐述算法思想）：首先令i=0，j=n-1,看arr[i]+arr[j]是否等于Sum，如果是，则结束。如果小于Sum，则i=i+1;如果大于Sum，则j=j-1。这样只需要在排好序的数组上遍历一次，就可以得到最后的结果，时间复杂度为0(N)。

伪代码如下：

|  |
| --- |
| for(i = 0; j = n - 1; i<j)  if(arr[i] + arr[j] == Sum)  Return(i, j)  else if (arr[i] + arr[j] < Sum)  i++;  else  j++;  return(-1, -1) |

寻找2拆分算法见has2sum()，拆分过程如step3所示，描述step3的函数为exchangeFunc()。

### 2.2.5 比较函数与最大交易次数控制

本问题下主要存在三种类型的比较：

1. 初始排序，对冲抵消额度相等的元素，函数为compItemNoMaxWeight
2. 拆分抵消过程中，产生的余额按照优先级以及余额比较插入，函数为compItem
3. 所有元素处理完成后，将结果按照初始顺序输出，比较标记原始顺序的数（非必要），函数为compOrder

其中初始排序直接按照额度排序即可，函数为compItemNoMaxWeight，函数名含义为不考虑优先级的情况下比较元素。按照原始数据输出，直接比较其标记原始顺序的值就可以。

对于拆分过程中，若存在2拆分，则直接记录交易过程，并将其存入相应存储容器就行；若不存在2拆分，则需要将Step3中的抵消后的余额插入到原容器中，插入位置的查找为二分查找。为了控制一个元素（一位打款人或一位收款人）的交易次数，其每交易一次其权重加1，当其交易次数weight大于等于max\_weight-1(max\_weight为最大权重，也即设定的最大交易次数)时，其在集合中的顺序将依据其权重来排序，权重越大，元素越靠后，将优先被交易。当其交易次数低于max\_weight-1时，其顺序将依据其余额来排序。

CompItem函数定义如下：

|  |
| --- |
| bool compItem(const Exchange\_Item& Item1, const Exchange\_Item& Item2)  {  if (Item1.getWeight() >= (Item1.maxWeight() - 1)) //Item1 已达到最大次数-1  { // Item2 已达到最大次数且超越Item1  if (Item2.getWeight() > Item1.getWeight())  return true;  else //Item2.getWeight() <= Item1.getWeight()  {  if (Item2.getWeight() == Item1.getWeight())  return compItemNoMaxWeight(Item1,Item2);  else  return false;  }  }  else//Item1 未达到最大次数  {  if (Item2.getWeight() >= (Item2.maxWeight() - 1))  return true;  else  return compItemNoMaxWeight(Item1, Item2);  }  } |

上述策略在其权重较低时不考虑其权重，使其能有最大的可能被抵消或是2拆分，这一点保证了全局呈现一个较优的结果；当其权重较大快达到最大权重时，为了避免其产生太多的交易次数，优先将其处理掉，这一点保证了一个元素被交易的次数不会太多。

### 2.2.6 算法时间复杂度分析

整个算法过程伪代码如下，语句后列出了大致的算法复杂度:

|  |
| --- |
| ExContainer Rers;//元素为收款人的容器  ExContainer Ters;//元素为打款人的容器  ExSaver RersSaver;//元素为完成交易的收款人的容器  ExSaver TRersSaver;//元素为完成交易的打款人的容器  readCSVdata(Rers, Rers\_file, 1);// 从CSV入数据  readCSVdata(Ters, Ters\_file, 1);  dealRepetition(Rers, Ters);//排序+对冲抵消 O（NlogN）+O(N)  Rers.sort(compItemNoMaxWeight);//再排序，完成交易的元素余额为0，排前面 O（NlogN）  Ters.sort(compItemNoMaxWeight);//再排序，完成交易的元素余额为0，排前面 O（NlogN）  toSaver（Rers，RersSaver）//将完成交易的元素存入RersSaver并从Rers中删除 O(N)  toSaver（Ters，TersSaver）  while (Rers.size() && Ters.size())//O（N）  {  if (Ters.back().getResMoney() > Rers.back().getResMoney())  exchangeFunc(Ters, Rers, TersSaver, RersSaver);//其中包括has2sum和交易记录,二分插入操作  else  exchangeFunc(Rers, Ters, RersSaver, TersSaver);// O(N)+O（logN）  }  RersSaver.sort(compOrder);//排序，按照原始顺序  TersSaver.sort(compOrder);//排序，按照原始顺序  writeResult(RersSaver, Rresult\_file);//写结果  writeResult(TersSaver, Tresult\_file);//写结果 |

分析以上算法不难得出，整个算法的时间复杂度由while循环部分决定，时间复杂度为。

# 算法测试

## 3.1算法正确性测试

手动编写两组数据，以验证算法的处理的“正确性”，产生如图3所示数据，其中左边为打款人信息，右边为收款人信息。



图3 验证样本（左为打款人信息，右为收款人信息）

算法处理结果如图4和图5。



**图4 打款人及其款项输出对象与金额**



**图5 收款人及其款项来源对象与金额**

分析以上结果可以看出，基本满足匹配需求的要求，且算法处理结果和手动分配结果基本一致。

## 3.2算法时间复杂度测试

未处理的对象以及处理完的对象均采用list容器时，算法测试结果如图6：



**图 6 算法测试结果**

其中1W样本为最开始给的样本数据集，40W样本为第二次给的样本数据集。选取样本按照向前选取，如40W样本的2W样本为，第二次的样本数据中，选取前20000个收款人，从打款人列表中选取前若干个人员，使其满足打款金额和收款金额相等，若不能满足相等，则从打款人中被选取的最后一个金额中取一部分，测试二中选取打款人31540位。

从图6可以看出，算法的时间复杂度基本为，这与之前的分析一致。

# 算法分布式实现想法与其他说明

## 4.1算法分布式实现想法

将数据集分为若干个小集合处理，如将10W的数据集分成10个1W，单核心顺序处理的话时间上只是增加10倍而不需要原来的200倍，采用10核心同时处理的话只增加同步以及通讯时间，整体时间略有增加。

但是不可将数据集分太小，将数据集分太小的话，数据分布不均匀，如出现一个特别大的金额，那这个数的交易次数会比较多，因此数据分割最好的是将数据分成分布比较均匀的情况，即有较大的金额的也有较小的金额，较小的金额充当凑数抵消之用。

## 4.2 其它说明

## 4.2.1 最大交易次数

最大交易次数，本解决方案中采用一个权重的方式，当其交易次数多时，增加其权重，提升其在排序中的优先级。

最大交易次数无法固定，如样本出现一个收款额是480000，而最大打款金额只有36000，因此收款方进行了16次交易。

合适的，足够大的最大交易次数，使得整体一致寻求更优的结果，会使得整体交易次数更少。

交易次数设定为5，也会出现部分用户交易出现6次或7次的现象，也就是说即使其交易了4次后，其会成为下一个要操作的对象，其本身也刚好是两个集合中最大的数，那样也至少还需要两次交易。

若想更大程度的限制一个用户的交易次数，可将最大交易次数减小，但是设置太小的话会出现大量的数均无法在设定交易次数内完成交易，均需要超最大交易次数完成交易，此时的最大交易次数的设定也就没有效果。

据样本的测试情况，平均交易次数一般小于2，也就是少，大多数1次交易就足够了，部分数据由于其本身的特殊性（本身金额较大，或者是对方存在大量的小金额）才会导致过多的交易次数。

经过测试，要控制最大交易次数，最好的办法是数据的分布均匀，保证打款和收款都有差不多的金额分布情况。（给的样本中，存在收款人普遍金额较大，打款人金额较小的情况）。

在最开始给的一个数据量差不多为10000的样本上做测试，设定不同最大权重结果如图7所示。最大交易次数设定为3时，最大打款次数降低了，但是总的交易次数却增加了。最大收款次数为16的出现是因为收款中出现一个非常的金额。



**图7 最大交易次数对结果的影响**

### 4.2.2 数据结构

数据结构，对时间花费有很大的影响。

在最开始的对冲抵消过程中，需要遍历元素，需要能实现快速随机访问

在step3中，由于存在中间插入删除操作，需要能满足高效率的插入删除操作。

若不能同时满足上述两个要求，需要权衡利弊，选择合适的。C++中选择list比选择vector要快很多。

# 参考文献

[1]子集和问题https://zh.wikipedia.org/wiki/%E5%AD%90%E9%9B%86%E5%92%8C%E5%95%8F%E9%A1%8C

# 附录

**子集和问题**

子集和问题与本问题有一定的相关，搜索子集和问题能找到较多的介绍资料，在此不赘述。

本方案中只需要用到2元素子集的问题，对于任意多元素子集和问题的处理算法和测试算法在SubsetSUM.cpp中给出。