Fast Getaway Loot Dividing

Reginald Lybbert, Kevin Tran, Sean Ovens, Paul Wang

April 8, 2016

Abstract

Robbing banks is fun, but only if you can get your share of the loot. In this paper we discuss an algorithm for dividing the loot equitably between two robbers.

1 Introduction

This section has yet to be written.

2 Problem Description

We have recieved the following problem description:

Two robbers come up with a scheme to rob a bank. Their getaway plan depends on dividing their loot as quickly as possible and then going their separate ways. Clearly, each robber wants an equal portion of the loot. As they grab items from the bank safe they stamp an estimated value on each item. They have asked you for an algorithm that will divide the loot as quickly as possible. The robbers are pretty sure that there is a way to divide the loot equitably but if this is not the case they have decided to abandon the venture altogether and leave all the loot behind.

Clearly these robbers are not highly intelligent, in that they should realize that even if they cannot divide the loot equitably, any division is better than leaving it all behind. However, including this assumption simplifies the solution well enough. Consider the following examples.

Example 1: Suppose our robbers manage to snag 5 items of worth $[1, 2, 3, 4, 10]$. We see that if one robber receives the item of value 10, and the other robber receives the rest, we see that both will have a total value of 10. Therefore, in this case there is a valid solution. Also note that it is the only valid solution. If one of the robbers receive the item of value 10, the only way to divide these items equitably is to give all of the remaining items (that sum up to a total value of 10) to his partner.

Example 2: It is possible that there are no valid solutions for an instance of the given problem. For example, consider the case where the bank being robbed only contains a single item of arbitrary (non-zero) value. In this case, neither robber can take this item, since there would be no items left for the other robber to take — thereby guaranteeing that there is no even split of the loot.

Example 3: It is also possible that there is more than one optimal solution to a given instance of the problem. Again, we can show this by example. Consider the case where the bank being robbed contains eight items, two with value 1, two with value 2, two with value 3, and two with value 4. In this case, one of the robbers can take items with values 1, 1, 4, and 4, with a total value of 10, while the other robber can take items with values 2, 2, 3, and 3, again with a total value of 10. Another optimal solution divides all eight items into two groups, each with a single item of each possible value (i.e. two groups containing items with values 1, 2, 3, and 4). Again, the total value taken by each robber in this circumstance would be 10, and thus this division constitutes another optimal solution to the problem.

It should also be noted that a valid solution to the problem is always optimal. Consider the total value of all of the items in a particular bank, and call this total value n. It should be reasonably clear that **every** valid solution to the problem will give each robber a set of items with total value $\frac{n}{2}$. If we increased the value that either robber received, we would unavoidably be decreasing the value that the other robber received. Thus, $\frac{n}{2}$ is not only the highest value that either robber can receive, but it is also the **only** value that either robber can receive. Thus, all valid solutions must also be optimal.

To develop an algorithm for this problem, we will need to carefully define our inputs and outputs. We desire an algorithm as follows:

Algorithm 1.1 [LOOTDIV]

Input: A set S of items, where each item consists of a unique ID, and a integer value,

Output: Two sets S_1, S_2 of items, each representing the share of a robber, or an error if this is impossible. Note $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$, and the sum of the values of the items in S_1 is equal to the sum of the elements in S_2.

---

3 Greedy Solution

Unfortunately, there does not exist a greedy solution to this problem, as stated. However, we can modify the problem, so that a greedy solution can be found. If we allow at most one piece of loot to be divided into pieces, we can produce a greedy algorithm.

One idea that we could use, would be giving the largest piece of remaining loot to the robber with the least loot currently allocated to him. However, this will not always produce a valid solution. For example, if $S = \{(a, 10), (b, 2), (c, 2)\}$, we would give $(a, 10)$ to robber A. At this point, robber A has 10, and robber B has 0. However, even if we give the rest of the loot to robber B, this we cannot make it even, as robber B could have at most 4.

Another idea that we could use, would be giving the smallest piece of remaining loot to the robbers on an alternating sequence, and splitting the last piece of loot to make it even. This idea should work, since we are trying to keep each robber's allocation as close to each other as possible. Later on, we will prove that it is always possible to split the last piece of loot to make the robber's shares even, which implies that this strategy always produces a valid, and optimal, solution. This strategy can be made precise as follows:

Algorithm 1.2 [Modified LOOTDIV]

Input: A set S of items, where each item consists of a unique ID, and a integer value. Each item is written in the form (a, v).

Output: Two sets S_1, S_2 of items, each representing the share of a robber, or an error if this is impossible. We must have either $S = S_1 \sqcup S_2$, or $S \sqcup \{(a_1, v_1), (a_2, v_2)\} = S_1 \sqcup S_2 \sqcup (a, v)$, $v = v_1 + v_2$, where $\sqcup$ denotes disjoint union, and the sum of the values of the items in S_1 is equal to the sum of the elements in S_2.

```
Let i = 1
While S has more than one element
      Take the item (a,v) from S with the smallest value
      Put (a,v) in set S_i
      Set i = 3 - i
End While
Let (a,v) be the last element in S.
Set j = 3 - i, d = |S_i| - |S_j|
Set v_1 = (v - d)/2, v_2 = v - v_1
Put (a_1, v_1) in S_i, and (a_2, v_2) in S_j
Output S_1 and S_2
```

Lemma 3.1 *Modified LOOTDIV always produces valid, and thus optimal, output*

Proof: Since the described algorithm will only divide the final (most valuable) element considered, if at all, it will suffice to show that the difference in value between the loot given to each robber before the final item is distributed is at most the value of this final element. Let $r_{a,i}$ denote the value of all items given to robber a after the ith step of the algorithm, where $a \in \{1, 2\}$ and $0 \leq i \leq n-1$ (with $|S| = n$). Also let the value of the final (most valuable) element be denoted by v_{max}. We proceed by using induction to show that $|r_{1,i} - r_{2,i}| \leq v_{max}$ for all $i \leq n-1$.

> *Note: On the* $(n-1)$*th step of the algorithm all items have been distributed except for the final item, whose value is not exceeded by any other element.*

Base case: $i = 0$. On the 0th iteration of the algorithm, no items have been distributed. Thus, $|r_{1,i} - r_{2,i}| = 0 \leq v_{max}$ since the value of each item is assumed to be non-negative.

Inductive hypothesis: Suppose that $|r_{1,i} - r_{2,i}| \leq v_{max}$ for all $i < k$, and show that $|r_{1,k} - r_{2,k}| \leq v_{max}$.

Suppose that $|r_{1,k-1} - r_{2,k-1}| = c$. Note that $c \leq v_{max}$ by the inductive hypothesis. On the kth step of the algorithm, we add the value of the least-valuable item available to us — call this value v_k. Note that $v_k \leq v_{max}$ due to the construction of the algorithm. The algorithm will proceed by offering v_k to the robber who has obtained the least valuable loot thus far. In other words, on the kth step of the algorithm we increase $min(r_{1,k-1}, r_{2,k-1})$ by v_k.

In the case that $r_{1,k-1}$ is the minimum, we have $r_{1,k} = r_{1,k-1} + v_k$, and $r_{2,k} = r_{2,k-1}$. Thus $|r_{1,k} - r_{2,k}| = |r_{1,k-1} + v_k - r_{2,k-1}|$. Now, note that $c = r_{2,k-1} - r_{1,k-1}$. So this equals $|v_k - c|$. Likewise, in the case that $r_{2,k-1}$ is the minimum, we have $c = r_{1,k-1} - r_{2,k-1}$. Thus $|r_{1,k} - r_{2,k}| = |r_{1,k-1} - r_{2,k-1} - v_k| = |v_k - c|$. Thus, in both cases, $|r_{1,k} - r_{2,k}| = |v_k - c| \leq v_{max}$, since $v_k \leq v_{max}$ and $c \leq v_{max}$, which is the required result. Therefore, when we consider the most valuable element in the input S, the difference in the values of each robber's loot is at most v_{max}.

Hence, the final five lines of the provided pseudocode will equate the loot taken by each robber, resulting in a valid solution. Since we have already seen that every valid solution to this problem is also optimal (refer to the argument on page 2), it follows that our greedy strategy is correct (i.e. all solutions provided by our algorithm are both valid and optimal).

□

Lemma 3.2 *Modified LOOTDIV runs in $O(n)$.*

Proof: The runtime function of the described algorithm is given by

$$T(n) = \sum_{i=1}^{n} c = cn \ ,$$

where c is some constant representing the cost of processing a single element of the input list of items. Since $T(n) \leq cn$ for some constant c and $n \geq 0$, $T(n)$ runs in $O(n)$ by definition of O.

4 Dynamic Programming Solution

4.1 Recursive Structure

Since we have failed in finding a greedy algorithm that solves this problem, we will now search for an algorithm using dynamic programming. We can start by dividing the problem into subproblems. Recall that we want to divide the loot into two equal piles. Consider the more general problem which divides the loot into a pile with a predetermined value. We see that this is a generalization of the original problem, since if we set the predetermined value to exactly half of the original value, we will get the loot for one of the robbers.

Now, we can use this as a way to describe subproblems for this. For example, if we have 4 pieces of loot, with values $\{5, 7, 12, 18\}$, we see that the total value is 42. Thus, we want to solve the problem of finding a pile with value at most 21. If we can find a pile with value 21, we can give that to Robber 1, and the remainder to Robber 2 in an equal division, but if such a pile does not exist, we cannot divide the loot equally.

We see that we have two choices. We can either include the object of value 18 or not. If we include it, we see that the remainder is a subset of $\{5, 7, 12\}$ with value at most 3. If we don't include it, we are lookinig for a subset of $\{5, 7, 12\}$ with value at most 21. These are both instances of the generalized problem that we are considering, thus we have a recursive structure for the problem.

We can formalize this structure as follows:

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,v) & \text{if } v_i > v \\ max(OPT(i-1,v), OPT(i-1, v-v_i) + v_i) & \text{if } v_i \leq v \end{cases}$$

In this structure, the parameter i represents the number of items being considered at a particular stage of the recursion. The parameter v denotes the value (i.e. half of the total value, in this case 21.) of items needed to complete the problem. Also note that v_i represents the value of item i (which could be any arbitrary item that is provided as input). The first case of our piecewise definition is our base case, that is, if $i = 0$ then there are no more items in the list to decide from and we return 0. In the second case, the call to $OPT(i-1, v)$ represents the decision to leave the ith item of the list to the other robber, and continue our search for value v in the remaining $i-1$ items. The last call to $OPT(i-1, v-v_i)+v_i$ represents the decision to take the ith item for ourselves, leaving us with $v-v_i$ total value to find in the remaining $i-1$ items. For instance, in the scenario described above the call to our recursive function would look like the following:

$$OPT(4, 21)$$

This call represents an instance of the problem that has access to 4 items, with a cumulative value that is 2×21. Intuitively, this function searches for the sum of any number of items that is as close as possible to the given v. Once found, this value is returned. Thus, when presented with an instance of the problem that has a valid solution, the presented recursive algorithm will return exactly v.

If we were to naively implement this without memoization, we would have an exponential time algorithm. We see this since in the worst case, we have two recursive calls at each level, where the input size only goes down by one. Thus the runtime recurrence is:

$$T(n) = 2T(n-1) + c_1 \quad T(0) = c_2$$

We claim that $T(n) \geq 2^n$. Since $T(0) = c_2 \geq 2^0$ applies for $n = 0$ this is our base case. Now suppose that $T(k) \geq 2^k$ for some $k \geq 0$. We can see that $T(k+1) \geq 2T(k) \geq 2(2^k) = 2^{k+1}$. Thus $T(k+1) \geq 2^{k+1}$ so $T(n) \geq 2^n$ for all $n > 0$. Hence $T(n) \in \Omega(2^n)$. Therefore, without memoization, this algorithm runs no better than exponential time.

Now, suppose that we use memoization. Then, we have a maximum of $n\frac{w}{2}$ calls, where n is the number of items, and w is the toal value of the items, as every call is of the form $OPT(i, v)$, where $0 \leq i \leq n$, and $0 \leq v \leq \frac{w}{2}$. Thus, if we use memoization, our worst case scenario is that the whole memoization table must be filled. This gives us a runtime of $O(nw)$, since each entry in the table can be computed in constant time, provided the previous columns have been filled. Thus, dynamic programming would allow us to reduce the time of this algorithm from exponential in n to polynomial in n and w.

4.2 Dynamic Programming Algorithm

Now that we have found a recursive substructure for this problem, we can develop a dynamic programming algorithm to solve it.

```
def solve(int values[n]):
   w = sum(values)/2
   initialize memo[n][w][3] to 0's

   def OPT(int i, int value):
          if (memo[i][value][2] == 1):
               return memo[i][value][0]
          else:
               if(i == 0):
                      memo[i][value][0] = 0;
                      memo[i][value][1] = 0;
               else if(values[i] > value)
                      memo[i][value][0] = OPT(i-1,value);
                      memo[i][value][1] = 0;
               else if(values[i] <= value)
                      include = OPT(i-1,value - values[i]) + values[i]
                      nInclude = OPT(i-1, value)
                      if(include < nInclude):
                            memo[i][value][0] = nInclude;
                            memo[i][value][1] = 0;
                     else:
                            memo[i][value][0] = include;
                            memo[i][value][1] = 1;
               memo[i][value][2] = 1;
               return memo[i][value][0];

     optValue = OPT(n,w)
     if(optValue != w):
          return None   //impossible to split the loot

     initialize s1,s2, as empty lists
     currValue = w
     for (int i = n, i > 0, i--){
            if memo[i][currValue][1] == 0:
                s2.append(i);
            else{
                s1.append(i);
                currValue -= values[i];
            }
      return (s1,s2)  //s1 are the indices for the items of robber 1, and s2 is the same fo
```

In this algorithm, we have a memoization table, called memo. This is most easily thought of as a 2d array, of 3-tuples. The first value of memo[i][v], is the optimal value of a subset of the first i items, with a total value less than or equal to v. The second value is 0 if this optimal subset doesn't include item i, and 1 otherwise. The third value is 0 if this value has not yet been computed, and 1 if the value is already in the table.

To analyze the runtime of this algorithm, we will need analyze the runtime of OPT. First, suppose that (memo[i][value][2]) is always zero, whenever we call the OPT algorithm. In this case, each call, with first argument i, calls at most two instances of OPT with first arguments $i+1$. Thus, OPT is called at most 2^i times. Since there are no loops in OPT, we see that this causes the runtime of OPT to be $O(2^i)$. We can do better than this though. We see that there are only nw entries in memo, thus as OPT only calls another instance of itself, if $memo[i][value][2]$ is still 0, or in other terms, that particular call of OPT hadn't been called before. Therefore, there are at most $2nw$ calls of OPT. Therefore, OPT is also $O(nw)$.

Now, if we look at the main algorithm, we make a single call to OPT, with an input of (n, w). The only other thing that may contribute to the runtime of the algorithm, is the for loop that will run linearly in n. Therefore, the call to OPT is the only major contributor to the runtime of solve. Thus, solve is $O(nw)$. Notice that this seems polynomial in the input, as the input is an array of size n that sums to $2w$, but we want the algorithm to be polynomial in the size of the input. Notice that w is bounded by 2^s, where s is the size of the data type used to store the values. Thus, as w is not polynomial in the size of the input, neither is $O(nw)$. So, solve is not a polynomial time algorithm.

5 Complexity Analysis

Originally, we considered a greedy strategy when approaching the proposed problem. Unfortunately, none of the obvious greedy strategies were capable of producing a correct solution to the problem in all cases. We continued by modifying the given problem slightly, before finding a polynomial time greedy solution to this new problem. This modification allows us to correct the "mistakes" that our algorithm made when choosing the items that would be given to each robber. However, the strategy that we used to solve the modified version of the Loot Division problem does not work for the problem as it is originally stated. The greedy strategy simply cannot consider enough information about the input at one time — there are no obvious ways to divide the original problem into the much simpler subproblems that are characteristic of greedy problems.

Next, we provided a recursive structure for the Loot Division problem. The recursive algorithm, when implemented naively, ran in exponential time. This is because, in the worst case, at a particular stage in the algorithm we would make two recursive calls while only reducing the input size of these calls by one. However, we realized that this implementation of the algorithm would solve identical problems repeatedly, and thus dynamic programming might be suitable for reducing the search space. For this reason, we developed a dynamic programming algorithm that utilized memoized recursion. With this technique, we achieved a runtime of $O(nw)$, where n was the number of items provided as input, and w was the total value of these items. This solution represented another failure to find a polynomial-time solution for the given problem, as the parameter w is not polynomial in the size of the input. Our attempt at finding an efficient solution to the LOOT DIVISION problem was again thwarted — this time because our dynamic programming solution was forced to recurse on cumulative item values.

The LOOT DIVISION problem can be rewritten as a decision problem as follows:

- Take as input a set of valuable items S and some integer v
- Return `yes` if and only if the set S has a subset with cumulative value v

Clearly, we can use the above decision procedure to find our whether a given instance of the LOOT DIVISION problem has a valid solution. One would simply need to call the above procedure with the input set along with half the cumulative value of the input set. We could also use the presented decision procedure to solve any instance of the problem as follows:

1) Query the decision procedure with the input set S and half the cumulative value of S
2) If the decision procedure returns `no` on its first invocation, then this particular instance of the problem does not have a valid solution
3) Otherwise, remove one item s from the set and query the decision procedure with $S - s$ (leave the value of the parameter v unchanged throughout this process)
4) If the decision procedure returns `no`, then place s back into the input set
5) Otherwise, grant this item s to the second robber
6) Repeat from step 3) (choose a new s) until every item has been considered at this stage

It should be reasonably clear that the above algorithm runs in linear time (disregarding the runtime of the decision procedure) since it considers each element of the input set S only once.

The LOOT DIVISION problem seems to be a numerical problem. We make this conclusion based on its similarity to the SUBSET SUM problem. In fact, the aforementioned decision procedure is exactly the same as the statement of the Subset Sum problem. Since the Subset Sum problem is classified as a numerical problem, the Loot Division problem must be a numerical problem as well.

We want to show that Loot Division is an NP-complete problem, thus we will first show that it is NP, and then give a reduction from the Subset Sum to Loot Division. Since Subset Sum is a well known NP-complete problem. This will prove that Loot Divison is also an NP-complete problem.

Lemma 5.1 Loot Divison *is NP*

Proof: To show that a problem is NP, we must find a certification algorithm, such that a certificate exists for every yes-instance of the problem. In this case, we will use the subset of items that are to be given to one of the robbers as the certificate. We now have the following certification algorithm:

```
def certify(int[] S, int[]S1):
     int sum;
     int total = sumValues(S);
     for (item,value) in S1:
           sum += value
           if item not in S:
                  return false;
           S.remove(item)
      end for
      if 2*value = total:
              return true;
     else:
             return false;
```

Notice that all of these operations can be performed in at most linear time, and the only loop traverses the elements of S1 exactly once. Thus the run-time of this algorithm is $O(n^2)$, assuming that *S.remove*(*item*) is a linear time algorithm.

Now, we must show that there exists a certificate for every yes-instance, but not for any no-instance. Suppose that it is possible to divide the loot evenly. Then, we can find a subset of S such that the sum of the values of the items in it are exactly half of the sum of the Values of the original set. In this case, we can pass in this subset. Since it is a subset of S, we will never enter the `if item not in S` branch, and the final value will be exactly half of the total value. Thus the certifier will return True.

Now, suppose that there is no such subset. Then, for the algorithm to return True, we must give is a set of items, all of which are in S, without repeats. (thus it is a subset), and with a total value of exactly half the initial value. However, by assumption, no such subset exists. Therefore, the certifier will return False.

Thus, since a certifier for Loot Division exists, we see that Loot Division is NP

□

Lemma 5.2 Subset Sum $\leq_p$ Loot Division

Proof: Consider the Subset Sum Problem. This problem is defined as follows:

Let S be a set of positive integers. Is there a subset of S such that it adds up to some value k?

We can reduce this problem to our loot division problem with the following algorithm:

```
def transform(int S[], int k):
     let n = sum(S);
     if k == n/2:
          return LootDiv(S);
     else if k > n/2;
           let S' = S.append(2k - n);
           return LootDiv(S');
    else if k < n/2;
           let S' = s.append(n - 2k)
           return LootDiv(S');
```

We see that this algorithm would run in polynomial time if we had a polynomial time algorithm for LootDiv, as it ca transform the input in a constant number of operation. Now, we must show that this reduction is correct:

We let n be the total of all the values of S, and we have three cases. Note that the value of a subset is the sum of its elements.

Case 1: $n = 2k$. In this case, we see that we are looking for a subset of S worth exactly half of the total. Thus, the complement of this subset with respect to S will also sum to k. Therefore, the set S will be divided equally. Note that this is exactly what our LootDiv algorithm calculates. Therefore, if we pass in S to LootDiv, we will get a yes, as there is a partition of S into two equal subsets. On the other hand, if there is no subset of S worth k, LootDiv will be unable to partition S into two equal subsets, as each of those would consist of a susbet that adds to k. Therefore, in this case, LootDiv will return ”yes” if and only if it is a yes-instance to the Subset Sum problem.

Case 2: $n < 2k$. In this case, we are looking for a subset of S worth more than half the total. We can pad the set S with an element of size $2k - n$, so that our desired subset will be exactly half of the total. Now, we see that we are looking at case 1. Suppose that there is a subset of value k in S. Then, since

$S \subset S'$, we see that there is a subset of value k of S'. Likewise, if there is a subset of value k in S', then either it includes the pad or it doesn't. If it doesn't include the pad, we have found a subset of size k in S, and if it does, then its complement in S' is a subset of value k in S. Therefore, from the arguments in Case 1, we see that the solution to LootDiv(S') is the same as the solution to SubsetSum(S,k).

Case 3: $n > 2k$. In this case, we are looking for a subset of S worth less than half of the total. If we instead look for a set of size $n - k$, we see that it's complement will be the solution to the original problem. Also, the complement of the original solution will be a subset of size $n-k$. Therefore, SubsetSum(S, k) = SubsetSum(S,n-k). Thus, since $n < 2n-2k$, we see that this reduces to case 2, where we look for a subset worth $n-k$. Thus $SubsetSum(S, k) = LootDiv(S')$, where $S' = S.append((2n - 2k) - n)$.

Therefore, this algorithm correctly solves the Subset Sum problem in polynomial time, given a polynomial algorithm for LootDiv.

Therefore, Subset Sum $\leq_p$ LootDiv

□

Theorem 5.3 Loot Division *is NP-complete*

Proof: From the previous two lemmas, we see that Loot Division is NP, and that some NP-complete problem reduced to it. Therefore, Loot Division is NP-complete.

□

6 Conclusion

While we have been able to develop progressively more effective solutions to the Loot Division problem, our attempts to achieve a polynomial-time solution have failed. Our proof that the given problem is NP-complete, provided in the previous section, suggests that further pursuits of a polynomial-time solution would likely be a waste of effort. The fact that the Loot Division is NP-complete means that its solution is at least as difficult to compute as all other known problems in the class NP. Effectively, this means that we will never develop an efficient procedure to solve the problem. Thus, we must either modify the problem or settle for an inefficient solution.

If we decide that modifying the original problem is acceptable, we can develop a greedy solution that runs in linear time. This runtime should be sufficient for most situations, and is certainly more manageable than an exponential-time solution. However, the modification that was made to make this solution possible may not be desirable — our greedy solution to this modified version of the problem does not necessarily tell us anything useful about the solution to the original problem. If we decide that modifications to the problem are not acceptable, then we may use the dynamic programming approach that was described in section 4 of the paper. Unfortunately, this means accepting a solution that does not run in polynomial time. This approach may be desirable if certain guarantees can be made about the input to the problem. Recall that our dynamic programming solution ran in a so-called ”pseudo-polynomial” time — meaning that while it did not run in polynomial time in the input size, it did achieve a polynomial runtime based on some other feature of the input. In the case of our dynamic programming solution, our proposed algorithm ran in polynomial time based on the cumulative value of all items provided as input. Thus, if we can be certain that we will never be given an input list that has a large cumulative value, then our dynamic programming solution is acceptable.

Whether one decides to use a greedy or dynamic programming approach when presented the LOOT DIVISION problem depends entirely on the requirements of the user. If modifications to the problem are acceptable, then a linear-time greedy solution is an attractive choice. Otherwise, one may be forced to use an exponential-time dynamic programming solution, which may itself not be desirable if we cannot make guarantees about the cumulative sums of the input lists. Due to the NP-completeness of the LOOT DIVISION problem, sacrifices must be made in the search for an efficient solution.