CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean
Webpage: `http://www.cs.clemson.edu/~bcdean/`
Handout 13: Homework #3

Fall 2018
TTh 12:30-1:45
Earle 100

1 Iterative Refinement and Recursive Search

This homework will give you some familiarity with the design and implementation of greedy algorithms, optimization methods based on iterative refinement, and pruned recursive search to exhaustively enumerate all the potential solutions of a problem.

Since this assignment is due right after Halloween, for fun we will describe it using a Halloween theme. The $n = 16$ lines of input in the file

`/group/course/cpsc212/f18/hw03/candy.txt`

describe n pieces of candy. Each line describes a single piece of candy in terms of its weight (in grams) and its value (tastiness). Ideally, you would like to collect all n pieces of candy. Unfortunately, however, you only have three bags that each can hold at most 2 kilograms — any more weight than this and a bag will break! Each piece of candy you decide to carry must be placed entirely in one bag; it cannot be cut up and placed partially in multiple bags.

Your goal in this problem is to determine the maximum amount of tastiness you can pack in total into all three bags, without any of the bags breaking. Despite our somewhat whimsical motivation, this problem, known as the *multiple knapsack problem*, is actually quite important in practice. For example, imagine we want to optimally pack rail cars or allocate tasks among different workers. The problem is also quite computationally challenging, being NP-hard, so obtaining an exact optimal solution for large n is computationally infeasible. In our situation, each of our n pieces of candy can be in one of 4 possible states: bag 1, bag 2, bag 3, or not present. This gives $4^n = 4,294,967,296$ possible solutions, some feasible and some not (e.g., some of these potential solutions may involve overfull bags). We could of course check all of them to find the best answer, but this would take a long time – probably several minutes.

2 First Approach: Greedy

One easy and extremely fast way to obtain a reasonable solution is to use a greedy algorithm, much like with the standard knapsack problem we discussed in class. The only difference with this problem is that we fill the bags sequentially, greedily adding items to bag 1 until just before it would overflow, then adding to bag 2, and so on. Your program should print on its first line of output the total value obtained by a greedy solution:

```
Greedy: 9762
```

Your solution should match this value, assuming you use the same greedy ordering as with the standard knapsack problem. For full credit, you need at least this much value.

3 Second Approach: Iterative Refinement

To generate better solutions, we turn to the idea of iterative refinement, using an approach much like with the traveling salesman lab:

```
For each of T iterations:
  Start with a random assignment of candy to bags
  Iteratively refine this assignment until it becomes locally optimal
Output the best value of all T final solutions
```

The larger you set T, the slower your program but the more likely you will find good solutions. For our purposes, please set $T = 1000$.

The tricky part of doing iterative refinement here is how to explore the neighboring solutions of a particular assignment. For example, given a particular assignment, you could generate neighboring solutions by, say, looking at all possible ways of moving a piece of candy from one bag to another. However, if you move a piece of candy from bag x to bag y, then bag y may now be full beyond its capacity, and bag x may now have room to fit additional candy. You may therefore need to "repair" your neighboring solution after the move by greedily removing candy from y (until it no longer overflows) and greedily filling unused candy into x (until right before it would overflow). This sort of "repair" procedure is common in iterative refinement situations where neighboring solutions might be slightly infeasible. Alternatively, you might want to regard all candy as being assigned to bags 1, 2, or 3 (i.e., no candy is left out), so the bags are always in an overfull state, and this is nice since it alleviates any feasibility concerns when moving candy around to obtain neighboring solutions. However, when assessing the value of a solution, we would only "count" the value of a greedily-chosen subset of items in each bag that fits within its capacity. There are several ways to think about defining and exploring neighboring solutions here, and you are encouraged to try other alternatives.

One final issue to consider is that many neighboring solutions may have the same value. For example, if you move an item from bag x to bag y and make no other changes, then this new solution will have the same value as the original solution. Do we move to this new solution? Even though it has the same value, perhaps it may look better in that one of the bags may now have much more free space than in the old solution, making it more likely you will be able to fit more candy in the future[1]. You may want to consider this issue when deciding on how to break ties between equal-value solutions.

The second line of output printed by your program should be the best solution value obtained via iterative refinement:

```
Refinement: 10591
```

For full credit, you should produce a solution of at least this value.

[1] Of course, the total free space will be the same. However, it might be preferable to move all the free space to a single bag, maximizing the chances of adding a new piece of candy.

4 Final Approach: Pruned Exhaustive Search

For the final part of this assignment, you will search through all possible solutions using recursive search, and much like in lab 8, you will prune away infeasible solutions early in the search to speed things up. Recall that there are 4 options for each item: bag 1, bag 2, bag 3, or unused. You should recursively try each of these options for all items in sequence, pruning your search whenever you have reached an infeasible solution. That is, you first try all possible settings for item #1, then for each of these you recursively try all possible settings for item #2, and so on. You may want to exploit symmetry to help reduce the running time; for example, it does not matter if the first item goes in bag 1, bag 2, or bag 3, since we could have just re-named the bags to make these partial solutions equivalent.

The third and final line your program prints should be the output of its pruned exhaustive search:

```
Exhaustive: 10658
```

For credit on this part, your program will need to obtain the number above. For full credit, your entire program needs to run in less than 10 seconds on the lab machines. Substantial partial credit will be given for a running time less than 20 seconds, and slightly less credit for a running time less than 1 minute. If the running time of your exhaustive search code is more than 1 minute, it will not likely receive much credit.

5 Submission and Grading

Final submissions are due by 11:59pm on the evening of Thursday, November 1. No late submissions will be accepted.