CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2021
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MWF 9:05-9:55
Handout 13: Homework #3 — Powers 208

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 soon after Halloween, for fun we will describe it using a Halloween theme. The $n = 16$ lines of input in the file

`/group/course/cpsc212/f21/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 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, called the *multiple knapsack problem*, is 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).

2 First Approach: Greedy

One fast way to obtain a reasonable solution is to use a greedy algorithm: consider the candies in decreasing order of tastiness divided by size, and for each candy, try to add it to any bag with sufficient capacity (there are several variants on this theme – feel welcome to experiment). Your program should print on its first line of output the total value obtained by a greedy solution:

```
Greedy: 9762
```

For full credit on this part, you need at least this much value.

Solution Representation. An issue you'll also need to decide for this part and the other two later parts of the assignment is how to store an assignment of candy to bags in memory. Two natural suggestions include a "candy centric" approach, where you store for each candy the bag containing it, or a "bag centric" approach, where you store for each bag the set of candies in it.

3 Second Approach: Iterative Refinement

To generate better solutions, we turn to the idea of iterative refinement, repeatedly making small changes to improve a solution:

```
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 to at least 100.

To explore the "neighboring" solutions of a particular assignment, try moving every piece of candy into a different bag, checking if this improves your solution. Of course simply moving a candy from bag x to bag y won't change the total value of a solution, but this may open up additional space in bag x into which you can then fit more formerly-unassigned candies. More generally, 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, so feel welcome to experiment with alternatives).

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.

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 or clearly sub-optimal 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 (this is optimal, so if your program prints something larger it must have a bug.)

Timing will be important for the score your code receives. For 100% credit, your *entire* program (all 3 parts) needs to run in less than half a second on the lab machines. Partial credit for slower programs will be granted, proportional to speed. For example, programs running in less than 1 second will receive only a minor deduction in points, programs running in less than 2 seconds will receive a more substantial deduction, and so on. Programs running in more than 5 seconds will receive at most 50% credit, and those running in more than 10 seconds will receive little credit.

5 Submission and Grading

Final submissions are due by 11:59pm on the evening of Wednesday, November 10. No late submissions will be accepted.