CS3100: Algorithms
Homework 8

1. You are asked to consult for a business where clients bring in jobs each day for processing. Each job has some processing time t_i that is known when the job arrives. The company has a set of 10 machines, and each job can be processed on any of those machines.

   Up until now, the company has been running the simple "minimize the makespan" approximation algorithm that we covered in lecture, which (as you will recall) always gives a 2-approximation. They've been told this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of the performance. However, they are reluctant to change the scheduling, since it is simple and jobs can be assigned to a processor as soon as they arrive.

   In particular, once you told them the makespan could be twice as large as optimal, they decided to analyze their performance and found a surprising fact: They have been running it each day for a month, and they have not observed it to produce a makespan more than 20% above the average load, $\frac{1}{10} sum_i t_i$.

   To try to understand why they never hit a factor-of-two behavior, you ask a bit more about the kinds of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, so that $1 \leq t_i \leq 50$ for all i, and that the total load $\sum_i t_i$ is quite high each day - always at least 3000.

   Prove that for these types of jobs, the makespan greedy approximation algorithm from class will indeed always find a solution whose makespan is at most 20 percent above the average (and hence optimal possible) load.

2. Now, you have been asked to act as a consultant for the Port Authority of an ocean-side city. They're currently doing good business, and their revenue is constrained almost entirely by the rate at which they can unload the ships arriving in their port.

   Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold up to K units of weight. (You can assume the w_i's and K are integers.) You can stack multiple containers in each truck, as long as you don't exceed total weight K on any one of them; the goal is the minimize the total number of trucks needed. (Note: This problem is NP-Complete, but you don't need to prove that fact.)

   A greedy algorithm (which should look familiar) for this might proceed as follows: Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until the next container would overflow the capacity K. Now declare this truck loaded and send it off, and start loading the next truck. This algorithm, by considering trucks only one at a time, might not get the best total packing.

   (a) Give an example of a set of weights and a value of K where this algorithm does not use the minimum number of trucks.

   (b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

3. Suppose you are given a set of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B: $\sum_{a_i \in S} a_i \leq B$. The sum of the numbers in S will be called the *total sum* of S. Your job is to select a feasible subset S of A whose total sum is as large as possible.

   (a) Consider the following algorithm:

   ```
   Let S ← ∅
   Let T ← 0
   For each i ← 1 . . . n
     If T + a_i ≤ B
         S ← S ∪ {a_i}
         T ← T + a_i
   return S
   ```

   Give an example where the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A. (In other words, this is NOT a 2-approx.)

   (b) Now, give a polynomial time approximation algorithm of this problem with the following guarantee: It returns a feasible set whose total some is at least half as large as the optimal feasible set. (In other words, find a 2-approximation.)

   Hint: Your algorithm should be at most $O(n \log n)$ time, but you can probably do better.