Greedy Algorithms

For each of the below problems, think about how you would design a greedy algorithm to solve it optimally. What are some possible options for the greedy choice you make at each step? Prove that there is always an optimal solution that makes the greedy choice, and prove that the problem exhibits the optimal substructure property (an optimal solution to the problem consists of the greedy choice and an optimal solution to the resulting subproblem). We will discuss some of these problems in class next week.

1. Making change. When you go to the grocery store and pay in cash, chances are you receive some change. Usually there are several possible ways of making change: if I need 37 cents in change, I could be given three dimes, a nickel, and two pennies, or I could be given a quarter two nickels, and two pennies, or I could be given 37 pennies, etc. But nobody likes carrying around lots of spare change (unless you have coin-operated laundry, perhaps) so we want the number of coins given as change to be as few as possible. In the *change making problem*, we have the following:

- **Input:** A value $V < 100$, in cents.
- **Output:** A set $\mathcal{S}$ of U.S. coins (quarter=25¢, dime=10¢, nickel=5¢, penny=1¢) such that the total value of the coins in $\mathcal{S}$ is equal to V and $|\mathcal{S}|$ is minimized (the set contains as few coins as possible).

2. Scheduling to minimize lateness. In the interval scheduling problem, we had a single resource (our supercomputer) that could only run one job at a time. Many users submitted jobs that they wanted to run, each of which had a fixed time at which the job must start and finish running. Our job was to come up with a schedule consisting of some subset of the jobs, such that there were no conflicts between jobs and the number of jobs we ran was maximized.

In the *scheduling to minimize lateness problem*, we still have a single resource that can only run one job at a time. However, there are some differences from our original setup. First, we are required to run all of the jobs. Given that some jobs might have conflicts, in order to make this possible our jobs now have more flexibility. In particular:

- All jobs are available to start running any time after time 0.
- Each job j has a deadline $d(j)$. This is the time by which the job would like to finish running, but we are allowed to let it finish later.
- Each job j has a size $x(j)$. This is the amount of (contiguous) time for which the job must use the resource.
- Each job j has finish time $f(j)$. This is the time at which the job is done using the resource, and it depends on the schedule we choose.
- Each job j has a lateness $\ell(j)$. This is the time by which the job violates its deadline: $\ell(j) = (f(j) - d(j))^+$. The notation $(y)^+$ means "take the positive part of y": that is, if $f(j) - d(j)$ is negative, the lateness is 0.

For example, if we have a job j with arrival time $a(j) = 2$, deadline $d(j) = 6$, and size $x(j) = 3$, we could start running the job at time 3, in which case its finish time would be $f(j) = 6$ and the deadline is violated, so the job has lateness $\ell(j) = 0$. We could also start running the job at time 5, in which case its finish time would be $f(j) = 8$ and the deadline is violated with lateness $l(j) = 2$.

In the scheduling to minimize lateness problem, our goal is to come up with a schedule that gives start times $s(j)$ for all of the jobs $j \in J$, and that minimizes the maximum lateness across all jobs. We have the following:

- **Input:** A set of jobs J, where for each job $j \in J$ we are told the arrival time $a(j)$, the deadline $d(j)$, and the size $x(j)$.
- **Output:** A schedule $\mathcal{S}$ of start times $s(j)$ for all jobs $j \in J$ that satisfies the following:
  - For each job j, $s(j) \geq a(j)$.
  - At most one job is using the resource at any given time.
  - For each job j, $f(j) = s(j) + x(j)$ (that is, the job is scheduled for a contiguous stretch of time).
  - The maximum lateness $L_{\max} = \max_{j \in J} \ell(j)$ is minimized.

3. Fractional knapsack. Imagine you are a thief and you are trying to steal some items. Unfortunately you can't take them all because you have a backpack that can only hold up to W pounds. There are n total items that you're interested in, and each item i has some positive integer weight w_i. Additionally, each item has a positive integer value v_i, and you would like to steal as much value as you can, while remaining within the weight constraint of your backpack. You are allowed to take a fractional amount p_i of an item (think of the items as being something like gold powder, rather than gold ingots). For the *fractional knapsack problem*, we have:

- **Input:** A set of n items, where each item i has a weight w_i and a value v_i, and a total capacity W that is the maximum weight your backpack can hold.
- **Output:** A fraction $p_i \in [0, 1]$ for each item i that indicates what fraction of the item you are stealing. The total value of the stolen items in your backpack, $V = \sum_{i=1}^{n} v_i p_i$, should be maximized, subject to the constraint that the total weight in your backpack, $W_B = \sum_{i=1}^{n} w_i p_i$, must be less than the capacity of your backpack, W.

Challenge: What if the items were gold ingots rather than gold powder? That is, suppose that instead of being allowed to take a fractional amount of each item, we can either take *all* of the item or *none* of it. Does a greedy algorithm still work?