Greedy Algorithms

CLRS 16.1-16.2

Sometimes we can solve optimization problems with a technique called *greedy*. A greedy solution has some quick and shallow way to evaluate each option without building the whole recursion tree under each option; it picks the option that looks best according to this shallow evaluation, includes it in the solution, and repeats.

Greedy algorithms are somwhat intuitive (human nature?) and seem much easier than dynamic programming, and they are, until you try to prove that they are correct. Many times (most of the time?) they aren't.

Before we look at a specific example here are the similarities and differences between greedy and dynamic programming.

Dynamic programming:

- The problem must have the *optimal substructure property*: the optimal solution to the problem contains within it optimal solutions to subproblems. This allows for a recursive solution.
- Dynamic programming: Evaluate *all* options *recursively*, pick the best and recurse. An option is evaluated recursively, meaning all its options are evaluated and all their options are evaluated, and so on; basically to evaluate an option, you have to go through the whole recursion tree for that option.
- Dynamic programming is essentially smart recursion (recursion without repetition). The subproblems are stored in a table to ensure that they are computed at most once. Typically the number of different subproblems is small, but the recursive algorithm implementing the recursive formulation of the problem is exponential. This is because of overlapping calls to same subproblem. So if you don't use the table to cache partial solutions you incur a significant penalty (Often the difference is polynomial vs. exponential).
- Sometimes we might want to take dynamic programming a step further, and eliminate the recursion —- this is purely for eliminating the overhead of recursion, and does not change the $\Theta()$ of the running time. We can think of dynamic programming as filling the table bottom-up, without recursion.

The greedy technique:

- As with dynamic programming, in order to be solved with the greedy technique, the problem must have *optimal substructure*.
- The problems that can be solved with the greedy method are a (very small!) subset of those that can be solved with dynamic programming.

- The idea of greedy technique is the following: Instead of evaluating *all* options *recursively* and picking the best one, we use a quick way to pick what looks locally like the best choice, add it to the solution, and repeat. So basically a greedy algorithm picks the *locally* optimal choice at each step, hoping to a solution that is *globally* optimal.
- Coming up with greedy heuristics is easy, but proving that a heuristic gives the optimal solution is tricky.

Let's look at an example.

Interval scheduling (Activity Selection)

Problem: Given a set $A = \{A_1, A_2, \cdots, A_n\}$ of n activities with start and finish times (s_i, f_i), $1 \leq i \leq n$, find a maximal set S of non-overlapping activities.

- Practical interpretation: corresponds to scheduling the maximal number of classes (given their start and finish times) in one classroom. Or more exciting: get your money's worth at Disney Land! you are in the park, you have lots of rides to chose from, you know start and end time for each ride, you want to ride as many rides as possible. Take Algorithms!
- This can be solved with dynamic programming: Look at the first activity, try a solution with it and one without it, and see which one is better. We can make it run in $O(n^2)$.
- Note that with dynamic programming we evaluate recursively both options (current activity is in or out), and pick the best. With a greedy solution, we would find a quick way to pick one or the other option.
- Here are some possible greedy algorithms for activity selection:

  Greedy 1: Pick the shortest activity, eliminate all activities that overlap with it, and recurse.

  Clearly all we need to do is sort the activities, so this would run in $O(n \lg n)$ time. Does this work? Yeah it works in the sense that it comes up with a set of activities that are non-overlapping, but is this set of activities optimal (i.e. largest set possible)? We need to argue that this algorithm always gives an optimal solution; or we need to give a counter-example (an instance where this strategy does NOT give the optimal solution).

  In this case it is fairly simple to find a counter-example:

- Greedy 2: Pick the activity that starts first (eliminate all activities that overlap with it, and recurse).

  Counter-example:

- Greedy 3: Pick the activity that ends first (eliminate all activities that overlap with it, and recurse).

  Counter-example:

A Greedy solution for Activity Selection

Turns out that picking activities in order of their finish time gives the correct optimal solution. We'll argue below why. The intuition is that by picking what ends first, we maximize the remaining time.

First let's spell out the idea a bit more:

1. Sort activity by finish time and renumber the activities so that $A_1, A_2, \cdots, A_n$ denotes sorted sequence.
2. Pick first activity A_1.
3. Compute B = set of activities in A that do not overlap with A_1.
4. Recursively solve problem on B.

Correctness

Before we think about running time, we need to argue that this algorithm is correct. Greedy outputs a set of activities. 1. Are they non-overlapping? 2. Is this set the largest possible? It's easy to see that 1 is true. For 2, the crux of the proof is the following:

Claim: Let A_1 be the activity with earliest finish time. Then there exists an optimal solution that includes A_1.

Proof: Suppose O is an optimal solution (a non-overlapping subset of A of max size).

- If $A_1 \in O$, we are done
- If $A_1 \notin O$:
  - Let first activity in O be A_k
  - Make new solution $O - \{A_k\} + \{A_1\}$ by removing A_k and using A_1 instead
  - This is valid solution (because $f_1 < f_k$) of maximal size ($|O| - 1 + 1 = |O|$

So this tells us that the first greedy choice is correct. What about the second one?

Once the first greedy choice is made, the problem reduces to finding an optimal solution for the remaining problem (the activities that don't conflict with A_1); this is the optimal substructure of the problem, which can be proved by contradiction.

The second activity chosen by the greedy aolution is the activity that finishes first in the remaining problem (activities that don;t conflict with A_1), so applying the claim again to the remaining problem we know there must exist an optimal solution that includes this second greedy choice.

And so on, it follows that at every step greedy stays ahead. There exists an optimal solution that consists entirely of greedy choices.

Implementing the greedy idea

The greedy idea above can be implemented in quadratic time: Sorting takes $O(n \lg n)$ time; step 2 takes $O(1)$ time; step 3 can be implemented in $O(n)$ time, and in step 4 we may recurse on $n-1$ activities, so overall this can be implemented in $O(n^2)$ time.

However, if we are a little more careful, once the activities are sorted, we can implement the greedy idea in $O(n)$ time. The crucial observation is that the solution consists of greedy choices that are compatible with previous greedy choices; so we pick the first activity A_1, then we traverse $A_2, A_3, ..$ in order, and discard all those that conflict with A_1; we pick the first activity that does not conflict with A_1. And so on. This leads to the following algorithm:

- Sort A by finish time.
- Schedule the first activity
- Then go through remaining activities in order and schedule the first activity that starts after the previously scheduled activity finishes.
- Repeat.

```
GreedyActivitySelection (A[1..n])
    Sort A by finish time
    S = {A_1}
    j = 1
    FOR i = 2 to n DO
        IF s_i ≥ f_j THEN
            S = S ∪ {A_i}
            j = i
```

Analysis: Running time is $O(n \lg n) + O(n) = O(n \lg n)$.
Example: Trace the algorithm on the following 11 activities (already sorted by finish time):

$$(1,4), (3,5), (0,6), (5,7), (3,8), (5,9), (6,10), (8,11), (8,12), (2,13), (12,14)$$

Comments, etc

- To be complete, a greedy correctness proof has three parts:

  1. Prove that there exists an optimal solution which contains the first greedy choice.
  2. Prove optimal sub-structure.
  3. Induction on the solution size to prove that there exists an optimal solution that consists entirely of greedy choices.

  Part 2 and 3 are usually omitted (same for all problems). To prove that a greedy algorithm is correct it suffices to prove that there exists an optimal solution which contains the first greedy choice.

- A greedy algorithm chooses what looks like best solution at any given moment; its choice is "local" and does not depend on solution to subproblems. (Greediness is shortsightedness: Always go for seemingly next best thing, optimizing the present without regard for the future, and never change past choices).

- In theory, we are only interested in greedy algorithms that are provably optimal. Most optimization problems cannot be solved greedily. Or put differently, the number of problems that have greedy solutions is very small (so chances are that if you came up with a greedy algorithms its probably wrong).

- The greedy technique can be applied to pretty much any optimization problem and is very popular in AI. There one deals with exponential problems or infinite search spaces, and one cannot solve these problems optimally, so the expectations are different. To emphasize that they are not optimal they are usually referred to as greedy *heuristics*.

- It is often hard to figure out when being greedy gives the optimal solution! Problems that look very similar may have very different solutions.

Example:

- $0-1$ KNAPSACK PROBLEM: Given n items, with item i being worth \$ v_i and having weight w_i pounds, fill knapsack of capacity w pounds with maximal value.

- FRACTIONAL KNAPSACK PROBLEM: Same, but we can take fractions of items.

  - FRACTIONAL KNAPSACK can be solved greedily:
  - Compute value per pound $\frac{v_i}{w_i}$ for each item
  - Sort items by value per pound.
  - Fill knapsack greedily (take objects in order)
  - Runs in $O(n \log n)$ time, easy to show that solution is optimal.