Greedy Algorithms

CLRS 16.1-16.2

Today we discuss a technique called "greedy". To understand the greedy technique, it's best to understand the 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.
- The idea of dynamic programming is the following: At every step , evaluate *all* choices *recursively* and pick the best. A choice is evaluated recursively, meaning all its choices are evaluated and all their choices are evaluated, and so on; basically to evaluate a choice, you have to go through the whole recursion tree for that choice.
- The table: The subproblems are "tabulated", that is, they are stored in a table; using the table to store solutions to subproblems means that subproblem are computed only once. Typically the number of different subproblems is polynomial, 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 sub-problems solutions 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 the *optimal substructure property*
- The problems that can be solved with the greedy method are a subset of those that can be solved with dynamic programming.
- The idea of greedy technique is the following: At every step you have a choice. Instead of evaluating *all* choices *recursively* and picking the best one, pick what looks like locally the best choice, and go with that. Recurse and do the same. So basically a greedy algorithm picks the *locally* optimal choice hoping to get the *globally* optimal solution.
- Coming up with greedy heuristics is easy, but proving that a heuristic gives the optimal solution is tricky (usually).

Like in the case of dynamic programming, we will introduce greedy algorithms via an example. We'll see the greedy technique in action through an example.

Interval scheduling /Activity Selection

One can think of this problem as corresponding 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 teh 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!!!

The less exciting name: interval scheduling or 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.

- This can be solved with dynamic programming: It's a special case of Weighted Interval Scheduling with all weights equal. The textbook has a different solution that runs in $O(n^3)$.
- For each activity, we have the choice to include it or not. In the dynamic programming solution, we evaluate recursively both choices, and pick the best. With a greedy solution, we would find a "quick" way to pick one or the other.
- Example: Here is a possible greedy algorithm for this problem: 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. What about correctness? We need to argue that this algorithm always gives an optimal solution; or we need to give a counter-example. In this case it is fairly simple to find a counter-example, that is, an instance where this strategy does NOT give the optimal solution.

  Counter-example:

- How about a different greedy algorithm: Pick the activity that starts first. Does this work?

  NO! Counter-example:

- How about a different greedy algorithm: Pick the activity that ends first. Does this work?

A Greedy solution: 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:

- Sort activity by finish time and let $A_1, A_2, \cdots, A_n$ denote sorted sequence.
- Pick first activity A_1
- Compute B = set of activities in A that do not overlap with A_1.
- Recursively solve problem on B.

Sorting takes $O(n \lg n)$ time, and steps 2,3,4 can be implemented in $O(n^2)$ time. If we are more careful, once the activities are sorted, we can implement the greedy idea in $O(n)$ time. The 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 the first that does not conflict with A_1, and so on:

- Sort A by finish time.
- Schedule the first activity
- Then schedule the next activity in the sorted list that comes 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):

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

Correctness: Output is set of non-overlapping activities, but is it the largest possible? 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? Well, the second interval chosen is the first interval in the remaining problem, so applying the claim again to the sub-problem we know there must exist an optimal solution that includes it. And so on, it follows that at every step greedy stays ahead.

For the mathematically inclined: 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. Justify that once a greedy choice is made (A_1 is selected), the problem reduces to finding an optimal solution for remaining problem (activities non-overlapping with A_1):

   Claim: Let O be an optimal solution for A containing A_1. Then $O' = O - \{A_1\}$ is the optimal solution for $A' = \{A_i \in A : s_j \geq f_1\}$.

   Proof:

   - Suppose by contradiction that O' is not optimal solution to A'; this means the optimal solution of A', call it O", has more intervals than O': $|O"| > |O'|$.
   - But $O" + \{A_1\}$ would be solution to A, of size $|O"| + 1$, which would be larger than $|O'| + 1$, which is the size of O. But this means we found something better than our presumed optimal solution O —- contradiction.
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 (usually they are the 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.*

Comments, etc

- 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. In many practical situations there is no choice choice (the optimal algorithms are too slow), and greedy algorithms are used even if they are non optimal. They are usually referred to as greedy *heuristics*.
- Greedy technique can be applied to 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.
- 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.