Lecture Notes for CSCI 311: Algorithms

Set 11-Introduction to Greedy Algorithms

Professor Talmage

March 4, 2026

---

1 Paradigm

We continue to explore recursive algorithms for optimization problems, but now turn our attention to a third approach for solving such problems: *Greedy Algorithms*. Since they are recursive solutions, we still require optimal substructure. Instead of saving repeated work like Dynamic Programming, though, Greedy Algorithms work by repeatedly making a choice *without* looking down the recursion to ensure it is optimal. Instead, we use only locally-available information to make a choice, then the algorithm makes any recursive calls needed to complete the solution. This is typically much faster than even Dynamic Programming, which makes many recursive calls to check all possible solutions.

Of course, for a greedy algorithm to generate an optimal solution, there must be extra conditions on the problem. Specifically, we have to prove that there is some optimal solution which contains all of the greedily-chosen elements. This is known as the *Greedy Choice Property*. Greedy algorithms are, by nature, often very simple, so proving the GCP (and optimal substructure) is often the bulk of the work for a greedy solution. We must also be careful in our choice of greedy strategy, as not every greedy choice leads to a correct algorithm, just like not every recursive breakdown led to a correct Dynamic Programming solution.

2 Example: Activity Selection

We start with an example problem, that of choosing a set of activities to fill available time. We must assign a particular time to each activity, and we must ensure that no two selected activities overlap. Picture this as allocating some fixed resource, such as a conference room, qPCR machine, SEM, compute server, Hubble, JWST, CERN, etc. The goal is to allow as many activities as possible, not necessarily the "most valuable".

2.1 Problem Statement

Input: $S = \{a_1, \ldots, a_n\}$, each $a_i = [s_i, f_i)$ (half open interval)

Output: Maximal subset $S' \subseteq S$ s.t. no two $a_i, a_j \in S'$ overlap

Exercise: Solve the problem for input

$$S = \{[7, 24), [10, 18), [15, 29), [13, 25), [16, 23), [28, 30), [3, 20), [6, 16), [9, 12), [3, 5)\}$$

2.2 Optimal Substructure

First, we need to understand how to break the problem down recursively. We cannot just consider arbitrary smaller time intervals, as which smaller intervals of time are available to be scheduled depends on which activities we have already selected. Instead, suppose we have selected a specific activity and consider the remaining available time.

1. Let $S_{ij} \subseteq S$ be the set of possible activities which start after a_i ends and finish before a_j starts.
2. If this were the entire input, we could choose a maximal-size subset of those activities. Call A_{ij} such a optimal solution.
3. Assuming S_{ij}, and thus A_{ij}, is not empty, let a_k be any activity in A_{ij}.
4. Split A_{ij} into $A_{ik} \cup \{a_k\} \cup A_{kj}$. That is, A_{ik} is all the elements of A_{ij} which end before a_k and similarly for A_{kj}.
5. Suppose A_{ik} is not optimal. That is, there is another non-overlapping subset A'_{ik} of S_{ik} with $|A'_{ik}| > |A_{ik}|$.
6. Nothing in A'_{ik} can overlap a_k, or anything in S_{kj}, so $A'_{ik} \cup \{a_k\} \cup A_{kj}$ is a valid solution to S_{ij}, larger than A_{ij}.
7. This contradicts our definition of A_{ij} as optimal, so A'_{ij} cannot exist. A similar argument shows that A_{kj} is also optimal.
8. Thus, an optimal solution is built from optimal solutions to smaller problem instances, so we have optimal substructure.

2.3 Recursive Definition and Parameterization

Define $c[i, j] = |A_{ij}|$ for every $1 \leq i < j \leq n$. We have to check every activity to see if it is in the optimal solution, maximizing over the best solution using that activity, which gives the recurrence

$$c[i,j] = \begin{cases} 0 & S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} c[i,k] + 1 + c[k,j] & \text{otherwise} \end{cases}$$

With this recurrence, we see that we have an $n \times n$ subproblem space, which we could fill in $O(n^3)$ time, similarly to our Dynamic Programming solution for MCM.

2.4 Faster Approach

Finding the maximum at every level is fairly expensive. Can we avoid it? Dynamic Programming makes it more efficient to try the same things repeatedly, but can we avoid that repetition entirely? To do this, we will need to be able to discard some candidate solutions as clearly not leading to an optimal solution.

Exercise: What properties might suggest that you should take an activity without further consideration. Give at least 2-3 different ideas.

- Overlaps nothing else in S (does not disqualify any other activity)
- Earliest start (counterexample: one long task blocks all others)
- Fewest overlaps (counterexample: 4 consecutive, 3 overlapping first 2, one overlapping 2nd and 3rd, 3 overlapping last two)

- Shortest duration (blocks least time but may leave gaps, counterexample: overlaps two others, could take both instead)
- Earliest finish time (blocks least of available time)

Exercise: Try each of the above strategies on the following set of activities:

i	1	2	3	4	5	6	7	8	9	10
s_i	1	2	1	3	7	8	6.5	10	12	11
f_i	3	3	5	6	8	8.1	10	11	15	22

2.5 Pseudocode

```
1:  function GreedyActivitySelection(S)        ▷ Assumes S sorted in order of increasing finish time.
2:      n = |S|
3:      A = {1}                                ▷ A will be the solution
4:      j = 1                                  ▷ Last-selected activity
5:      for i = 2 to n do
6:          if s_i ≥ f_j then
7:              A = A ∪ {i}
8:              j = i
9:          end if
10:     end for
11:     return A
12: end function
```

Exercise: Work through the code on the following set of activities.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	13

Runtime: $\Theta(n)$ on sorted input. $\Theta(n \log n)$ if unsorted since we need to sort, then run this algorithm.

Correctness:

Theorem 1. *`GAS`(S) produces solutions of maximum size for the Activity Selection problem.*

Proof. Let $S = \{e_1, \ldots, e_n\}$ be a set of activities, $e_i = [s_i, f_i), \forall 1 \leq i \leq n$. Assume $f_i \leq f_j$ iff $i \leq j$. We will show that there is an optimal choice which contains e_1. Suppose $A \subseteq S$ is some optimal solution. Let e_k be the earliest-finishing activity in A. Note that e_k must finish before any other activity in A starts. If $k = 1$, we are done, as A is an optimal solution containing e_1. If $k \neq 1$, we will convert A into another optimal solution which does contain e_1.

Let $B = A \backslash \{e_k\} \cup \{e_1\}$. No two activities in B conflict, because $f_1 \leq f_k$ and $s_i \geq f_k$ for all $e_i \in A \backslash \{e_k\}$. Thus, B is a valid solution to the activity selection problem on input S, and $e_1 \in B$. $|B| = |A| - 1 + 1 = |A|$, so B is an optimal solution. □

Corollary 1. *$B \setminus \{e_1\}$ is an optimal solution to the subproblem with input $\{e_i \mid s_i \geq f_1\}$.*

Proof. By contradiction, adding e_1 to a better solution to the subproblem would form a better solution to the original problem (input S) would yield a solution better than B, which is a contradiction, since B is optimal. □

We thus conclude that the algorithm yields correct solutions, since it chooses the first-finishing activity (e_1), then discards any activities overlapping that, and repeats.

3 General Outline for Proving GCP

In any greedy algorithm, we have to prove the greedy choice property. We just did for Activity Selection, though we did not call it that explicitly. Thankfully, these proofs typically follow a specific format:

1. Consider some globally optimal solution.
2. Alter that solution to contain the first greedy choice.
3. Show that the resulting solution is of equal value to the optimal solution, and is thus also optimal.
4. Show that the remaining work is a subproblem.
5. Optimal Substructure implies an inductive proof that there is always an optimal solution containing the greedy choice.

4 Aside: Knapsacks and Input Size

Recall the Knapsack problem from recitations. We have discussed why always taking the item with highest value-density is not optimal for the 0-1 version of the problem.

Exercise: What was the fundamental problem with this greedy strategy? For what version of the Knapsack problem does this strategy give optimal solutions? What runtime does this give?

For the 0-1 problem, a Dynamic Programming solution uses the recurrence

$$maxKnap(W, S_i, V_i) = \max(maxKnap(W, S_{i-1}, V_{i-1}), v_i + maxKnap(W - s_i, S_{i-1}, V_{i-1}))$$

where $S_i = [s_1, \ldots, s_i]$. This considers each item (in reverse index order), and compares the maximum values achievable by either leaving or taking it. This recurrence gives a 2-dimensional subproblem table indexed by

- $1 \leq i \leq n$, indicating which items are still under consideration
- $1 \leq x \leq W$, indicating how much more weight we can carry

This leads to a $\Theta(nW)$ algorithm, since each cell is computed in constant time from two others. However, this problem is not known to be solvable in polynomial time. In fact, if it is, then a lot of problems we think are impractically expensive suddenly become feasible.

This seems contradictory, since nW appears to be polynomial (in two variables). The fact that we are multiplying two different variables is not the problem. Instead, using the *value* instead of the *size* of an input is a problem. For reasons beyond the scope of this class (but covered in CSCI 341: Theory of Computation), algorithm runtime is a function of input size, not input value.

The input to the Knapsack problem has size $n + \log_2 W$, since we will express the numerical value W in binary, which takes $\log_2 W$ bits. $W = 2^{\log_2 W}$, so the runtime nW is exponential in (part of) the input size.

We call such algorithms, which are polynomial in the value of numerical inputs, *pseudo-polynomial*. These, including the 0-1 Knapsack problem, are computationally very hard, meaning that it does not take a very large instance before it takes an impossibly long time to compute. The takeaway for us is that we need to be very careful with numerical inputs to code and their effects on runtime. Lists of elements are less of an issue, since the input size for a list of n elements is at least n, as each element appears in the input.