Chapter 7

Greedy Algorithms

With a series-of-choices approach, the solution is built up incrementally by making a series of choices or decisions. For example, consider the event scheduling problem:

> Given a collection of events with start time $s(i)$ and finish time $f(i)$ $(0 \leq s(i) \leq f(i))$, find the largest set of non-overlapping events.

One way to formulate this as series of choices is to consider each event in turn — for each, the choice is: include the event in the solution set, or not? But how do we decide which of those alternatives to pick for each event?

A *greedy* algorithm always picks the locally best alternative for each decision — that is, the choice is made based on the information present at the time the choice is being made and not on any consideration of future possibilities. This means that we're not concerned with whether making a particular choice now prevents a better option in the future — we just make the choice that seems to do the best job of achieving the goal now.

For the event scheduling problem, the goal is the largest set of non-overlapping events, so the best choice at the moment would be to increase the number of events picked if possible. Thus: include the current event in the solution set if it doesn't overlap any of the ones already chosen.

But does this strategy actually result in the largest possible set of non-overlapping events? Proving correctness is the crux of greedy algorithm development.

7.1 Greedy Is Not Always Good

It is not a given that a greedy algorithm can be found for a given problem, particularly for optimization problems. Problems for which a greedy algorithm *can* find an optimal solution have two key properties:

- **Greedy choice property**. The globally optimal solution can be found by making locally optimal choices.
- **Optimal substructure property**. An optimal solution to the problem can be constructed efficiently from optimal solutions of subproblems.

The second property is essential for being able to build up the solution one step at a time — without the optimal substructure property, it might be necessary to undo parts of an earlier solution-so-far in later iterations. For example, consider the problem of finding the shortest path graph — the optimal substructure property means that if B is on the shortest path from A to C, the shortest path from A to C includes the shortest paths from A to B and from B to C.

7.2 Elements of Greedy Algorithms

7.2.1 Main Steps

Since only one alternative for each decision needs to be considered, the series of choices can be implemented using a loop:

```
while not done
  make the next choice
```

7.2.2 Greedy Choices and Counterexamples

The heart of a greedy algorithm is how the choices are made — defining the criteria by which an alternative is selected. This must be a local choice, meaning that future consequences aren't considered.

To start with, ask: does the alternative picked matter? Most likely it does, and it is easy to find a counterexample to demonstrate that, but it's always good to make sure that there's not a simple solution before jumping into the work of finding a more complex one.

To identify possible greedy choices, consider what information is available (or could be computed) about each element and how that could be used to achieve a correct result. Keep the choices simple — elaborate conditions involving a combination of multiple pieces of information will likely be more difficult to prove things about — and plausible — you aren't arguing correctness yet, but if there's no reason to think that a particular greedy choice might result in the desired behavior, there's no reason to consider it further.

For example, the input for the event scheduling problem contains the start and finish times for each event ($s(i)$ and $f(i)$, respectively). From that the event duration $d(i) = f(i) - s(i)$ could be computed. There are thus six possible choices for which event to pick next: earliest starting, latest starting, earliest finish, latest finish, shortest, and longest. With the goal of finding the largest set of non-overlapping events, plausible options are shortest event first (shorter events are less likely to overlap, so greater potential to pick more non-overlapping events), earliest-ending event first (more time after that event for additional events), and latest-starting event first (more time before that event for additional events). One would not expect picking longer events to be advantageous in avoiding overlaps, and there's no reason to think that simply starting earlier or ending later would help either.

However, many plausible greedy choices are incorrect. Attempting to find counterexamples for each of the plausible greedy choices should eliminate most of the possible choices from further consideration — if all are eliminated, greedy is not the right approach for this problem. (More than one valid greedy choice is unlikely, but not necessarily impossible.)

7.2.3 Loop Invariants

For optimization problems, the loop invariant must address both *legality* — that the solution-so-far is valid (it doesn't violate the structural constraints of a solution) – and *optimality*.

The legality part takes the typical form for iterative algorithms: *After k iterations, the solution-so-far is still valid.*

For the optimality part, something stronger is needed. A common form is to use a *staying ahead* argument: After k iterations, our algorithm's partial solution is at least as good as any optimal solution after k steps.

The "at least as good as" criteria should be defined in terms of the greedy choice, not the optimization goal. For example, if the greedy choice for the event scheduling problem is based on the earliest ending time, the staying ahead argument should be defined in terms of the earliest ending time: the finishing time of the kth event picked by the algorithm is no later than the finishing time of the kth event in an optimal solution. Defining staying ahead in terms of the optimization goal leads to a trivial or nonsensical statement: After k iterations, our algorithm has picked at least as many events as any optimal solution has after picking k events.

It is important that the comparison between the algorithm's partial solution and an optimal solution be an apples-to-apples comparison. Also, if the output is a set (the ordering of the elements doesn't matter), it can be useful (and necessary) to consider the elements in the optimal solution to be ordered in the same way the algorithm considers them. For example, if the algorithm is picking events in order of finishing time,

we'll consider the events in the optimal solution in order of finishing time as well so that loop invariant is comparing the finishing times of the kth earliest finishing event in the algorithm's solution and in an optimal solution.

Because the measure by which the algorithm's partial solution is compared to an optimal solution is based on the greedy choice rather than the optimization criteria, the "final answer" step of showing correctness may not be as simple as it typically is with iterative algorithms. It can be useful to keep this step of the correctness proof in mind as you consider potential loop invariants — make sure that your loop invariant will be useful for showing the correctness of the final answer before spending a lot of time proving that it is maintained.

7.2.4 Final Answer

The loop invariant, based on the greedy choice, may not directly address the optimality of the result when the loop terminates.

For cases where optimality is based on the number of elements in the solution (such as the largest set of events), a strategy for this step is to consider the two unwanted possibilities: $|A| > |O|$ and $|A| < |O|$ where $|A|$ and $|O|$ are the number of elements in the algorithm's and in an optimal solution, respectively. One of those possibilities is impossible due to the definition of "optimal" — the algorithm cannot find a better solution than the optimal — and that the other also cannot occur is argued using the loop invariant, exit condition, and ending steps. If $|A| > |O|$ and $|A| < |O|$ are both impossible, it must be the case that $|A| = |O|$ and thus the algorithm produces an optimal solution.

7.2.5 Room For Improvement

Consider whether sorting the input elements according to the greedy choice criteria could speed up the implementation.

7.3 How to Design Greedy Algorithms

Greedy algorithms are iterative algorithms so the development process builds on the iterative template (section 3.3) with a few additions/modifications:

Establish the problem.

- *Specifications.*
  For optimization problems, also identify and distinguish between legal solutions and optimal ones.

Identify avenues of attack.

- *Paradigms and patterns.*
  Consider the patterns defined in section 6.2.
- *The series of choices.*
  Identify the series of choices (section 6.4.1).
- *Greedy choices and counterexamples.*
  Identify the criteria by which the greedy choice can be made, and attempt to find counterexamples for each in order to eliminate plausible-but-wrong greedy choices.

Define the algorithm.

- *Main steps.*
  This is the core of the algorithm — the loop body. For greedy algorithms, the main steps take the form

```
while not done
  make the next choice
```

Show termination and correctness.

- *Loop invariant.*
  State a loop invariant. The loop invariant often takes a form as outlined in section 7.2.3.
- *Final answer.*
  Explain why the whole algorithm — setup, loop, wrapup — means that the final result is a correct answer to the problem. This argument can take the forms outlined in section 7.2.4.

Determine efficiency.

- *Room for improvement.*
  Include consideration of whether sorting the input elements according to the greedy criteria can speed up the implementation.

That makes the full process as follows:

Establish the problem. Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*
  State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, also identify and distinguish between legal solutions and optimal ones.
- *Examples.*
  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack. Determine the algorithmic approach(es) to try. The approach and paradigm steps are intended for when a paradigm is not pre-determined; skip them if the paradigm is already decided.

- *Known targets.*
  Identify any applicable time and space requirements for your solution. This might be stated as part of the problem ("find an $O(n \log n)$ algorithm"), come from having an algorithm you are trying to improve on, or stem from the expected input size (e.g. you need to work with very large inputs so you need $O(n \log n)$ or better).
- *Approach.*
  Identify applicable approach(es): divide and conquer or series of choices? For each, consider briefly what that approach would look like for this problem — does it even make sense?
- *Paradigms and patterns.*
  Consider the series-of-choices patterns defined in section 6.2.
- *The series of choices.*
  Identify the series of choices (section 6.4.1).
- *Greedy choices and counterexamples.*
  Identify the criteria by which the greedy choice can be made, and attempt to find counterexamples for each in order to eliminate plausible-but-wrong greedy choices.

Define the algorithm. Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Main steps.*
  This is the core of the algorithm — the loop body. For greedy algorithms, the main steps take the form

  ```
  while not done
    make the next choice
  ```

- *Exit condition.*
  When does the loop end?
- *Setup.*
  Whatever must happen before the loop begins.
- *Wrapup.*
  Whatever must happen to get the final answer after the loop ends.
- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).
- *Algorithm.*
  Assemble the algorithm from the previous steps and state it.

  There shouldn't be new elements here, instead bring together the main steps, exit condition, setup, and wrapup along with any handling needed for special cases and state the whole algorithm.

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*
  Explain why the algorithm always terminates i.e. it always eventually produces a solution.
  - *Measure of progress.*
    Identify a quantity and the direction of change that leads towards the exit condition.
  - *Making progress.*
    Explain why every iteration of the loop advances the measure of progress towards the exit condition.
  - *The end is reached.*
    Explain why making progress ensures that the exit condition is always reached.
- *Correctness.*
  Explain why the solution produced is the correct solution for every valid input instance.
  - *Loop invariant.*
    State a loop invariant. The loop invariant often takes a form as outlined in section 7.2.3.
  - *Establish the loop invariant.*
    Explain why the loop invariant holds at the beginning of the first and second iterations of the loop.
  - *Maintain the loop invariant.*
    Explain why the loop invariant continues to be true after each iteration — assuming that it holds at the beginning of iteration k, explain why it also holds at the beginning of the next iteration $(k+1)$.
  - *Final answer.*
    Explain why the whole algorithm — setup, loop, wrapup — means that the final result is a correct answer to the problem. This argument can take the forms outlined in section 7.2.4.

Determine efficiency. Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.
- *Time and space.*
  Assess the running time and space requirements of the algorithm given the implementation identified.
- *Room for improvement.*
  Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues for improvement. Include consideration of whether sorting the input elements according to the greedy criteria can speed up the implementation.