Divide-and-conquer works by dividing the task into independent subproblems which are solved separately; the solutions from the subproblems are then combined in some way to get the overall solution.

A different strategy is to formulate the process of building a solution as making a series of choices. 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.

An algorithm to solve this problem (from the produce output perspective) could be summed up as follows:

```
initialize an empty set of events
repeatedly
  select a non-overlapping event
until no non-overlapping events remain
```

The choice here is which event to add to the solution set next, and the series of choices arises from repeatedly choosing a next event.

This problem can also be thought of from a process input perspective —

```
initialize an empty set of events
for each event
  if it doesn't overlap anything already in the set,
    decide whether or not to add it to the set (and add it if so)
```

In this the choice is deciding whether or not to include a particular event in the set, and the series of choices comes from making a decision about each event.

In both algorithms, the key question is *how* to make each choice — in the first algorithm there may be many non-overlapping events to choose from and in the second algorithm there are always two possible options (add or don't add). Related to this is *whether it is possible* to choose the right alternative on the spot, or if more than one alternative needs to be considered in order to find the desired answer.

Chapter 5

Greedy Algorithms

When the process of building a solution can be framed as making a series of choices, a key question is how to make each choice. A *greedy* algorithm always makes a local decision — that is, one alternative is picked 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 choosing a particular alternative 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.

Greedy algorithms often come up in optimization problems, where the goal is to find the best solution amongst many legal solutions, but the key characteristic — that the right alternative can be chosen with a purely local decision — is also relevant when the goal is simply to find a legal solution.

Because a single alternative is picked at each decision point, greedy algorithms are iterative algorithms, and our development process for greedy algorithms is based on the one for iterative algorithms.

5.1 Does Greedy Work?

It is important to realize that a greedy strategy doesn't always work — it is not always the case that the right alternative can be chosen with a purely local decision. 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.

(These properties can be adapted for non-optimization problems: a greedy algorithm can find a legal solution if a legal solution can be found by making locally valid choices, and a legal solution can be constructed from legal 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.

Identifying whether a particular problem satisfies these properties is, of course, essential for the ultimate success (or not) of a greedy algorithm for the problem but it isn't necessary to prove that greedy is applicable as the first step of developing a greedy algorithm — after all, the process of coming up with a successful greedy algorithm is also the process of demonstrating that the greedy choice property holds. However, it is worth considering the optimal substructure property — if you can find a counterexample, then you know that pursuing a greedy algorithm isn't going to be successful.

5.2 15.5 Steps to a Greedy Algorithm

Greedy algorithms are iterative algorithms, and the development process is largely the same as the 15-step iterative development process (section 2.1). However, there are a few additional elements specific to greedy algorithms.

Establishing the problem.

1. *Specifications.*
   For optimization problems, also explicitly identify the optimization goal (what does "best" mean?) and be sure to distinguish between optimal solution(s) and legal ones.

Brainstorming ideas.

5.5 *Greedy choice.*
The main steps involve repeatedly making a local decision. On what basis is that choice made?

(a) *Greedy strategies.* Identify plausible greedy strategies for making each choice.

(b) *Counterexamples.* Narrow down the candidates by looking for counterexamples to eliminate plausible-but-incorrect greedy strategies.

5.3 Greedy Flavors

"A series of choices" is a broad categorization, and can apply to many different kinds of tasks. However, there are some common "flavors" of tasks that come up in greedy algorithms, including:

- Select a subset of the input items.
- Order the input items.
- Assign labels to the input items.

In all cases there is also a constraint that limits what constitutes a valid solution, and, for optimization problems, a notion of "best" amongst the valid solutions.

Identifying a familiar flavor can help further define a framework for the algorithm when combined with the common iterative approaches.

task	iterative pattern	main steps structure
"select a subset"	process input	for each element, decide whether to include in the solution or not include in the solution
	produce output	repeatedly find elements to include in the solution
"order the input items"	process input	for each element, add to the solution in the proper order
	produce output	repeatedly find the next element in the ordered solution
"assign labels"	process input	for each element, determine its label

Table 5.1

5.4 Developing Greedy Algorithms

We now consider aspects of the iterative development process as it applies to greedy algorithms in more detail.

Example(s). Examples are important both to explain the task to be accomplished, but also in the context of counterexamples for potential greedy choices. Initially focus on clarifying the specifications (as needed), but examples may be added later as potential greedy choices are considered.

Targets. The brute force algorithm for "series of choices" problems is to try all possible combinations of choices, of which there will generally be an exponential number. (c^d where c is the number of alternatives for each choice and d is number of decisions made) Greedy algorithms typically allow for a polynomial-time solution.

Approaches. Identify the particular greedy flavor, if applicable, as well as the applicable iterative approaches and what they look like for this problem.

Greedy strategies. To identify potential greedy strategies, consider what information you have about each element and consider how you could use each piece of information as the basis for picking one element or making one choice instead of another. Keep it 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 solution, there's no reason to consider it.

Counterexamples. Narrow down the plausible greedy strategies by looking for counterexamples to eliminate ones that aren't going to work.

Main steps. Combine the main steps structure shown in table 5.1 with a plausible greedy choice.

It is worth noting that the "select a subset" process input and produce output approaches are related to each other — if the elements are sorted according to the choice criterion, "repeatedly find elements to include in the solution" amounts to the same thing as "for each element in order, decide to take or not to take".

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

A direct statement — that what we want to be true for the entire solution holds for the part completed so far — still often works for the legality part of the invariant.

However, unlike the typical "process input" pattern, the optimality part of the loop condition is *not* that we have the optimal solution for the first k items. Instead, two common forms are

- *Staying ahead*: our algorithm's partial solution after k steps is at least as good in some respect as any optimal solution after k steps
- *We haven't gone wrong yet*: our algorithm's partial solution is still consistent with an optimal solution for the whole problem

For a "staying ahead" argument, a specific goodness measure is needed — the "at least as good as" language is a template and needs to be defined for the particular algorithm. The goodness measure is a quantity that can be compared, and it should be stated whether "at least as good as" means a bigger quantity or a smaller quantity. The particular measure depends on the problem, and is not always the same quantity that is being optimized — in fact, the goodness measure may be more closely connected to what the greedy choice is based on that the optimization criteria.

For both forms of invariant, if the output is a set (and thus the ordering of the elements doesn't matter, just what they are), it can be useful (and necessary) to consider the elements in the optimal solution ordered the same way the algorithm considers them. That allows for an apples-to-apples comparison.

Final Answer. Since it can take some effort to show the "maintain the loop invariant" step, it can be a good strategy to first test whether it's worth proving the invariant correct — can a particular invariant be used to show the correctness of the final answer?

The goal of this step is still to show that the loop invariant being true at the end of the final iteration combined with the exit condition and any wrapup steps means that the final answer is correct.

For loop invariants that have a simple "process input" form — the task is to maximize X, and the invariant is that after k interations, the algorithm's value of X is as big as or bigger than the optimal's value of X — this step is straightforward. (The exit condition means that $k = n$ when the loop exits, so the invariant holding yields that after (all) $k = n$ iterations, the algorithm's value of X is as big as or bigger than the optimal's value of X — and since the definition of "optimal" means that the algorithm can't have found a better solution, the algorithm's solution is optimal.)

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 possibilities when the loop terminates: either $|A| > |O|$ or $|A| < |O|$ or $|A| = |O|$ where $|A|$ and $|O|$ are the number of elements in the algorithm's and in an optimal solution, respectively. To show that it must be the case that $|A| = |O|$, argue why the other alternatives are impossible: $|A| > |O|$ means that the algorithm found a larger set than the optimal solution, which is impossible due to the definition of "optimal", while why $|A| < |O|$ is impossible must be shown using the loop invariant, exit condition, and wrapup steps. Then, 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. (Note that this is assuming a maximization goal; the cases are reversed for a minimization goal.)

Maintain the loop invariant. For a staying ahead loop invariant, proof by contradiction is often a good strategy: assume that the invariant is true after k iterations but is no longer true after $k+1$, and argue why that cannot happen.

Note that it is *not* valid to attempt a direct argument by saying that the invariant is maintained because the algorithm's process (namely the greedy choice) is correct. Whether or not the algorithm's process is correct is what we are trying to show — the overall correctness of the algorithm is established indirectly via the argument that because the invariant holds, the algorithm's process must be the right process. As a result, it is necessary to show that the steps the algorithm takes lead to the loop invariant holding.

5.5 Example

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, distinguish between legal solutions and optimal ones. output:

> 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.
>
> Input: n events with start and finish times $s(i)$ and $f(i)$
>
> Legal input: $0 \leq s(i) \leq f(i)$
>
> Output: a set of events
>
> Legal solution: a non-overlapping set of events
>
> Optimization goal: largest set

Examples. If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

No explanatory examples are needed; we might need to identify some examples as counterexamples for greedy choices later.

Targets. What are the time and space requirements for your solution?

No constraints on time or space are given in the problem. Trying all combinations of sets of events is exponential time, so we are aiming for polynomial time. Sublinear is not likely to be possible because it is possible that none of the events overlap and thus the solution contains all n.

Tactics. The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

Polynomial time means not looking at every possible combination of sets of events — a successful greedy strategy will achieve that so there's not really anything to say here.

Approaches. Identify the particular greedy flavor, if applicable, as well as the applicable iterative approaches and what they look like for this problem.

This is a "select a subset" problem.

Process input: For each event, decide whether or not to include it in the set.

Produce output: Repeatedly choose events to include in the set.

Greedy strategies. Identify plausible greedy strategies for making each choice.

For process input, a plausible greedy choice would be to include the event if it doesn't overlap with events already included in the solution — this is plausible because more events are better, so we include an event if we can.

For produce output, the greedy choice is about which event to choose next. Available event properties are length, starting time, and finish time — one could choose event in order of shortest or longest, earliest starting or latest starting, or earliest finish or latest finish. Of these, shortest (because there's more time remaining for other events to fit into), latest starting time (because there's more time before for more events to fit into), and earliest finish time (because there's more time after for more events to fit into) are plausible options for achieving the goal of most events selected. There's no reason to think that picking longer, earlier-starting, or later-finishing events would help us fit in more.

Counterexamples. Narrow down the candidates by looking for counterexamples to eliminate plausible-but-incorrect greedy strategies.

- *Include the next event if it doesn't overlap with events already included in the solution.*
  The problem here is that a long event chosen early could block lots of shorter events not yet considered. For example, event A runs from 2pm to 6pm, event B runs from 1pm to 3pm, and event C runs from 4pm to 8pm. The largest non-overlapping set would contain events B and C but an include-if-no-overlap strategy with the events considered in the order A, B, C means that A would be included, blocking both B and C.
- *Take the shortest remaining non-overlapping event next.*
  We look for a short event that nonetheless overlaps several others. For example, event A runs from 2-4pm, event B runs from 11-2:30pm, and event C runs from 3-8pm. Event A will be chosen first since it is shortest, but then A blocks the other two events. Choosing B and C is the optimal solution.
- *Take the remaining non-overlapping event with the latest starting time next.*
  For a counterexample, we want to set up a case where event A has the latest starting time but choosing it blocks both B and C, neither of which overlaps with the other (so B and C is the optimal solution). Both B and C must then start before A starts so that A has the latest starting time, and must end after A starts in order for them to overlap A. But then

both B and C must span A's start time, overlapping with each other. Perhaps there is a more complex counterexample, but the failure of one counterexample gives some additional plausibility to this as a potential greedy choice.

- *Take the remaining non-overlapping event with the earliest finish time next.*
  For a counterexample, we want to set up a case where event A has the earliest finish time but choosing it blocks both B and C, neither of which overlaps with the other (so B and C is the optimal solution). Both B and C must start before A ends and end after A ends in order to overlap with A and not be chosen before A. But this means that both B and C span A's finish time and thus overlap with each other. Perhaps there is a more complex counterexample, but the failure of one counterexample gives some additional plausibility to this as a potential greedy choice.

Main steps. This is the core of the algorithm — the main loop, focusing on the loop body. What's being repeated?

Counterexamples were found for several of the greedy choices identified, and we can observe that latest-starting and earliest-finishing are really just mirror images of each other — the difference is whether the solution is built up with later events first or earlier events first. Since going from earlier to later is perhaps a bit more natural an ordering, we'll go with earliest-finishing as our greedy choice.

Also, while this is a greedy choice associated with a "produce output" pattern, recall that for "select a subset" problems, there's an equivalent "process input" version where the elements are first sorted according to the choice criterion and then whether or not to take each element is considered in turn. Since it is often easier to think in terms of process input, make that switch.

```
for each event (in order of increasing finish time)
  choose the event if it doesn't overlap any already-selected events
```

Exit condition. Identify when the loop ends.

The process input pattern is that the loop ends when all of the input items have been processed.

When all of the events have been considered. (Alternatively, when there are no non-overlapping events remaining.)

Setup. Whatever must happen before the loop starts (initialization, etc).

Sort the events in order of increasing finish time.

Initialize the set of selected events to empty. (Nothing has been selected yet.)

Wrapup. Whatever must happen after the loop ends to produce the final solution.

The set of selected events is the answer — no need to do anything else after the loop ends.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for those cases as needed.

Duplicates or ties can often cause problems, so that's a good thing to check.

Two events with the same finish time will get sorted into some order and one will be considered before the other — it doesn't cause any problems for carrying out the algorithm.

Whether same-finish-time events being considered in a arbitrary order causes problems for correctness will be considered in the correctness argument.

Algorithm. Assemble the algorithm from the previous steps and state it.

```
sort the events by finish time
initialize the set of selected events to be empty
for each event
  add it to the selected events if it doesn't overlap with anything already selected
```

Termination. Show that the loop eventually terminates.

These steps often come directly from the iterative pattern.

Measure of progress. What metric can be used to tell you that you are getting closer to the solution?

The measure for the "process input" pattern is typically the number of input elements considered.

> The number of events for which a take/no-take decision has been made.

Making progress. Explain how each iteration of the loop changes the value of the measure of progress.

> In each iteration one more event is considered and either included or not, increasing the number of events for which a take/no-take decision has been made by one.

Reaching the end. Explain why making progress means that eventually the exit condition will be satisfied.

> Every iteration makes a take/no-take decision about one event and no event is considered more than once — n will be reached eventually.

Correctness. Show that the algorithm is correct.

Loop invariant. A loop invariant is a boolean statement about the state at the start of the loop body.

For optimization problems, the loop invariant needs to address both legality and optimality. We try a staying ahead argument for the optimality part. Since the greedy choice is based on the earliest finish time, we define the staying ahead goal in terms of the earliest finish time.

> None of the selected events overlap, and $f(A_k) \leq f(O_k)$ where A_k is the kth event selected by the algorithm (which chooses things in order of increasing finish time) and O_k is the kth event an optimal solution (when ordered from earliest-finishing to latest-finishing).

The first part addresses legality — the set of events cannot contain overlapping events.

The second part addresses the optimality of the solution. (Recall that in the original problem specifications, $f(i)$ is the finish time of event i.) It is a staying ahead argument — since the greedy choice deals with the earliest-finishing event, we compare the finish time of the kth event selected by the algorithm and the kth event in an optimal solution with the idea that "being ahead" means an equal or earlier finish time. Note that since there isn't any inherent ordering of the events in the solution — a solution is a set of events — we consider the elements in the optimal solution in the same order as the algorithm i.e. ordered by finish time.

Establish the loop invariant. Explain why the loop invariant is true at the beginning of the first iteration, and why, if it is true at the beginning of the first iteration, why it is also true at the end of the iteration / beginning of the second iteration.

> For $k = 0$, no events have been selected yet so there can't be any overlaps amongst selected events and there aren't any finish times to compare.
>
> For $k = 1$, the algorithm has chosen a single event — the earliest-finishing event.
>
> - *None of the selected events overlap.* A_1 is the first event selected so there's nothing already-

chosen for it to overlap.

- $f(A_1) \leq f(O_1)$. Since the algorithm considers events in order of finish time and A_1 is the first event considered, A_1 is the earliest-finishing event. For the optimal solution, O_1 is earliest-finishing of the events in the optimal. However, since A_1 is the earliest-finishing of all the events, O_1 can't possibly end any earlier and $f(A_1) \leq f(O_1)$.

Maintain the loop invariant. Assume that the loop invariant is true at the start of an iteration, and explain why the invariant is still true at the end of that iteration.

Assume that the loop invariant is true when a loop iteration begins: the k events already selected by the algorithm do not overlap, and $f(A_k) \leq f(O_k)$.

On the next iteration, the next event is either taken or not taken.

Consider the "no-take" choice: If the next event considered overlaps with any events $A_1, ..., A_k$, it is not selected and k doesn't change. Hence the invariant still holds.

Consider the "take" choice: If the next event considered does not overlap with any events $A_1, ..., A_k$, it is selected as A_{k+1}.

- *None of the selected events overlap.* We need to show that event A_{k+1} does not overlap with any events $A_1, ..., A_k$ — which is true because only non-overlapping events are selected.
- $f(A_{k+1}) \leq f(O_{k+1})$. Assume the invariant is true up to this point: $f(A_k) \leq f(O_k)$. Now also assume that this is the step where things go wrong: $f(A_{k+1}) > f(O_{k+1})$. What can we conclude?

  First, note that events A_{k+1} and O_{k+1} must be different events since they have different finish times.

  Since A_{k+1} is non-overlapping and the algorithm considers events in order of finish time, event O_{k+1} must overlap with something the algorithm already picked or else it would have been chosen instead of A_{k+1} since it ends earlier than A_{k+1}. Overlapping with something $A_1, ..., A_k$ means $s(O_{k+1}) < f(A_k)$ — event O_{k+1} must start before event A_k (the latest-finishing event selected by the algorithm) finishes.

  But no events in an optimal solution can overlap, and since $f(O_{k+1}) \geq f(O_k)$ (optimal's events are being considered in order of finishing time), $s(O_{k+1}) \geq f(O_k)$. The loop invariant gives us that $f(A_k) \leq f(O_k)$ so $s(O_{k+1}) \geq f(A_k)$ — which flatly contradicts the requirement that $s(O_{k+1}) < f(A_k)$ from the previous paragraph.

  Thus the assumption that the algorithm went wrong in this step is at fault, and $f(A_{k+1}) \leq f(O_{k+1})$.

Final Answer. Explain why, with the loop invariant being true and the exit condition being false when the loop completes, the wrapup steps lead to a correct result.

We need to show that we have the largest set of non-overlapping events. The non-overlapping part comes directly from the first part of the loop invariant — none of the selected events along the way overlap, so none of the selected events in the final set overlap.

But optimality? The loop invariant only says something about finish times: $f(A_{|A|}) \leq f(O_{|A|})$, where $|A|$ is the number of events selected by the algorithm. Since this is a problem about the number of elements in the solution, consider the two unwanted possibilities:

- $|A| > |O|$. This is impossible, because the definition of the optimal solution is that it is the largest possible set of non-overlapping events. Since the algorithm generates a legal solution (it does not pick overlapping events), it cannot end up with a larger set of non-overlapping events than an optimal (also legal) solution.
- $|A| < |O|$. This means that there is at least one more event in the optimal solution than in the algorithm's solution. Since the events in the optimal solution are being considered in

order of finish time, $f(O_{|A|+1}) \geq f(O_{|A|})$. The optimal solution does not contain overlapping events, so this means $s(O_{|A|+1}) \geq f(O_{|A|})$ — and, because $f(A_{|A|}) \leq f(O_{|A|})$ by the loop invariant, $s(O_{|A|+1}) \geq f(A_{|A|})$. But if the extra event in the optimal solution starts after the algorithm's last event finishes, it can't overlap with any other events that the algorithm picked — so it can't exist or else the algorithm would have picked it. Thus this case is also impossible.

Since $|A| > |O|$ and $|A| < |O|$ are both impossible, it must be the case that $|A| = |O|$ and thus the algorithm produces an optimal solution.

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.

We need to sort and iterate through a collection of events, access start and end times for an event, and repeateadly check for overlaps and, potentially, add an event to a collection. All of these, except for checking for overlaps, are well-known operations in well-known data structures with well-known running times and no tradeoffs (i.e. we can achieve all the desired running times in the same data structure) so we don't need to spell out the details.

(We don't know the specific data structures for how the input is provided, but most collections support $O(1)$ per element traversal and property-access (such as getting an event's start time) is typically assumed to be $O(1)$ (accessing an instance variable of an object, lookup by index in an array, or lookup in a hashtable-based Map). Sorting is well-known to be $\mathrm{O}(n \log n)$ for an array, but if we have some other kind of collection, an array can be built for the cost of traversal, which is less than the time it takes to sort, so it can be safely assumed that sorting is $\mathrm{O}(n \log n)$ regardless of the specific collection data structure.)

Since we are considering events in order of increasing finish time, the last-selected event will have the latest finish time of any already selected. Thus we only need to check that the next event's starting time is no earlier than the last-selected event's finish time to know that there aren't any overlaps with any already-selected events. This is $\mathrm{O}(1)$.

Time and space. Assess the running time and space requirements of the algorithm given the implementation identified.

The beginning steps require sorting the n events ($\mathrm{O}(n \log n)$) and initializing an empty collection ($\mathrm{O}(1)$).

The loop repeats n times, and each iteration requires determining if an event overlaps any of the already-selected events and, possibly, adding the event to list. As observed in the previous step, the overlap check can be done with a single comparison of times ($\mathrm{O}(1)$). Adding to an unordered collection is also $\mathrm{O}(1)$ so the total running time of the loop is $\mathrm{O}(n)$.

The final runtime: $\mathrm{O}(n \log n)$ to sort + $\mathrm{O}(n)$ to select events = $\mathrm{O}(n \log n)$.