A sidebar indicates the "answer" for each step — it is content that is part of following the algorithm development process, separate from commentary about the process or the solution itself.

Captioning

Your company has been hired to add closed captioning to live television broadcasts. Given a list of programs to be captioned (and the start and end times of each program), assign those programs to your employees using as few employees as possible. Note that each person can only caption one show at a time.

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.

> Task: Assign programs to employees using as few employees as possible but no employee is assigned overlapping programs.
>
> Input: n programs, with start and end times
>
> Output: programs with the employee assigned to each
>
> Legal solution: all programs are assigned an employee and no employee has been assigned overlapping programs
>
> Optimization goal: minimum number of employees

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

> No constraints were given.
>
> The brute force algorithm is to enumerate all possible assignments, starting with $k = 1$ and increasing k until a solution is found. For each k, the number of possible assignments is k^n, then it takes $O(n)$ time to check each assignment to see if it is a legal assignment (no overlapping programs). This is exponential.

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 an "assign labels" task.
>
> Process input: for each program, assign an employee

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

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

> There are actually two choices here: which employee to assign to a given program, but also which program to caption next.
>
> The employee assigned must not already be assigned to an overlapping program; the simplest thing is to choose the first such employee found and only assign a new employee if none are available.
>
> Programs can be ordered by start time, end time, or length. Plausible would be earliest start time (so there aren't extra gaps between programs), earliest end time (same idea), and shortest first (doesn't tie up the employee for a long).

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

> Programs can be ordered by start time, end time, or length. Shortest first can be easily dismissed — in the example above, the programs would be considered in the order A, B, C, D, which has already been shown not to result in an optimal solution. Longest first also doesn't work:
>
> - A 10am-1:30pm
> - B 3:15-5:45pm
> - C 1-3pm
> - D 2:30-3:30pm
>
> A and B are assigned to employee 1, C to employee 2, and D to employee 3 but only two employees are needed if employee 1 is assigned to A and D and employee 2 is assigned to B and C.
>
> Earliest start time and earliest end time both work for these examples, and seem like they may work more generally.

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

> For each program in order of start time, assign an employee who is not assigned any overlapping programs or a new employee if there is no one available.

Exit condition. Identify when the loop ends.

> When all of the programs have been assigned an employee.

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

> Sort the programs by earliest start time.

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

> Nothing else is needed.

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

> Multiple programs starting at the same time can be considered in any order — the "maintain the invariant" step requires only that already-assigned programs start at or before the time of the current program.

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

> `caption(programs)` —
>
> Input: n programs, with start and end times
>
> Output: programs with the employee assigned to each
>
> Legal solution: all programs are assigned an employee and no employee has been assigned overlapping programs
>
> Optimization goal: minimum number of employees

```
sort the programs in order of increasing start time
for each program,
  assign an employee who is not assigned to any overlapping programs,
   or a new employee if there isn't anyone available
```

Termination. Show that the loop eventually terminates.

> **Measure of progress.** The number of programs which have been assigned to an employee.
>
> **Making progress.** In each iteration, a program without an employee assigned is assigned to an employee.
>
> **Reaching the end.** With a new program assigned an employee in each iteration, eventually all of the programs will have been assigned an employee.

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. It purpose is to ensure that the algorithm remains on track towards the desired solution.

The loop invariant needs to address both legality and correctness of the partial solution. For greedy algorithms, the correctness part often takes the form of a staying ahead argument: after k iterations, the algorithm's partial solution is no worse than the optimal solution at that point. Since the order in which the programs are listed in the output doesn't matter, compare the programs the algorithm has labeled to those same programs in the optimal solution. ("The first k in the optimal solution" doesn't have any meaning because the order doesn't matter.)

> After k programs have been assigned an employee, no employee has overlapping programs and the number of employees used is no more than the number of employees assigned to those same k programs in the optimal solution.

Establish the loop invariant. Explain why the invariant is true after the setup and before the first iteration of the loop. Assuming the invariant is true before the first iteration, explain why it is still true after the first iteration and before the second.

The first step is to show that the loop invariant holds for both $k = 0$ (before the first iteration) and $k = 1$ (after the first iteration).

> $k = 0$. No programs have been assigned to employees so no employee has overlapping programs and both the algorithm and the optimal solution use 0 employees.
>
> $k = 1$. The program with the earliest start time has been assigned to an employee. That program must also be assigned to an employee in the optimal solution, so both the algorithm and the optimal solution use 1 employee. Only one program has been assigned to an employee, so no employee can have overlapping programs.

Maintain 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.

For the next step, assume that the loop invariant holds after k iterations, and show that it is still true after $k + 1$. Consider each part separately.

> *Non-overlapping programs.* The loop invariant gives us that no employee has overlapping programs amongst the first k programs, so if there's an overlap after $k + 1$ programs, it is because program $k + 1$ is assigned to an employee who already has an overlapping show. But programs are only assigned to available employees, so program $k+1$ can't have overlapped with something

already assigned to that employee.

No more employees than the optimal. The loop invariant gives us that after k programs have been assigned employees, the algorithm hasn't used any more employees than the optimal does for those same k programs. Assume that this is the point when things go wrong — after $k+1$ programs, the algorithm now uses more employees than the optimal does for those $k+1$ programs. If so, the algorithm must have assigned a new employee for program $k+1$ while the optimal did not — there are no values e and e' where $e \leq e'$ but $e+1 > e'+1$.

The algorithm assigns a new employee for program $k+1$ only if all of the others are already busy. Since the algorithm is considering programs in order of start time, those other k programs all started at the same time as or before program $k+1$ (the algorithm considers programs in order of start time) and they must end after $k+1$'s start time (in order to overlap with with $k+1$'s start time). That means there are $k+1$ programs at the same time and $k+1$ overlapping programs requires at least $k+1$ employees, so the optimal can't use fewer. This violates the assumption that this was the point where the algorithm fell behind (because that required the algorithm to use a new employee but the optimal not to), so it must be the case that the algorithm uses no more employees for its first $k+1$ programs as the optimal does for those same shows.

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.

For the final step, show that the loop invariant being true after the last iteration, plus the wrapup, results in the correct answer to the problem.

The exit condition is that that all programs have been assigned an employee; plugging $k = n$ into the loop invariant gives us that after n programs have been assigned to an employee, no employee has overlapping programs and the number of employees used is no more than the number of employees assigned to those same n programs in the optimal solution. Those n programs are *all* the programs, so we have a complete solution with no overlaps and no more employees than the optimal. Since the algorithm can't actually assign *fewer* employees than the optimal (if it did, the optimal solution wouldn't be optimal), it must assign the same number and thus be optimal.

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

It takes $O(n \log n)$ time to sort the programs at the beginning. The main steps loop repeats n times and it takes $O(e)$, where e is the number of employees, to find an available employee for each program. The total running time is thus $O(n \log n + en)$, or $O(n^2)$ since $e \leq n$.