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.

## Callers on Hold

***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.

> $n$ people are on hold, waiting for a single tech support operator. Let $t_i$ be the time it will take to solve person $i$'s problem. (The $t_i$ times are known in advance.) In what order should the operator handle the calls in order to minimize the total waiting time? (The total waiting time is the sum of the times each person has to wait until the operator answers her call.)
>
> Input: $n$ people with the time $t_i$ to handle person $i$'s call
>
> Output: an ordering of the $n$ people
>
> Legal solution: the ordering includes every caller exactly once
>
> Optimization goal: minimize total waiting time

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

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

> The brute force try-all-possible-orderings of people would be $\Theta(n!)$ — $n$ choices for the first call answered, $n-1$ for the second, $n-2$ for the third, etc.

***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.

***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 ordering problem.
>
> Process input: for each caller, determine where in the order to answer their call.
>
> Produce output: repeatedly find the next call to answer.

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

> The only information we have about callers is the length of the call $t_i$.

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

It may be easier to think about the produce output strategy, since the choice is directly which call is next.

> For produce output, the choice is which call to answer next. With the goal of minimizing the total waiting time, the only plausible choice would be to answer the shortest remaining call next — if a long call is answered, everyone else has to wait on that call. Putting long calls last means more people wait less.

For process input, the next call has to be placed in the ordering — based on the length of the call, with the shortest calls first.

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

Both approaches are based on the same strategy — shortest call first. It is probably easier to think about counterexamples for the produce output approach.

For a counterexample, we'll need a situation where answering a longer call next is better for the overall waiting time. But doing that means more people waiting on the longer call that was answered, and fewer on the shorter call that wasn't (yet) — it doesn't seem likely that we'll find a counterexample.

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

As noted already, both approaches involve the same choice. Let's try produce output as it focuses more directly on the choice itself.

```
repeatedly
  add the shortest remaining call to the ordering
```

***Exit condition.*** Identify when the loop ends.

All the calls have been added to the ordering.

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

The ordering is initially empty.

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

Nothing more to do — the ordering is the solution.

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

Consider things that might cause problems for the algorithm outlined so far.

If there are several calls with the same shortest length, pick any of them next.

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

> `answerCalls(t)` —
>
> Input: times $t$ for $n$ calls
>
> Output: an ordering of the calls
>
> Legal solution: the ordering includes every call exactly once
>
> Optimization goal: minimize total waiting time
>
> ```
> the ordering is empty
> repeatedly
>   add the shortest remaining call to the ordering
> ```

***Termination.*** Show that the loop eventually terminates.

The measure of progress should be directly related to the exit condition.

> **Measure of progress.** The measure of progress is the number of calls in the ordering.
>
> **Making progress.** Each iteration adds another call to the ordering.
>
> **Reaching the end.** A call is added to the ordering each time, so eventually they all will be.

***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 produce output loop invariant pattern says to try something of the form "After the first $k$ output elements have been generated..." This is an optimization problem, so we need to cover both legality and optimality of the solution in order to establish correctness. A direct statement ("we have an optimal solution so far") is often not successful, so we'll try the staying ahead pattern. In this, we compare the algorithm's progress at a certain point to an optimal solution at the same point. Since the goal is the minimum total waiting time —

> After $k$ calls have been added to the ordering, there aren't any repeated calls and the total waiting time for the algorithm's solution is no bigger than the total waiting time for an optimal solution.

Is this a good loop invariant? Think ahead to trying to establish the base case ($k = 1$) — no matter which call is answered first, the total waiting time for a solution with only one call is 0. If how the choice is made doesn't factor into the correctness of the invariant for the first choice, it's not likely we're going to have any basis for arguing that the invariant holds for later iterations. (Or the choice is irrelevant and the order in which the calls are answered doesn't matter. But it is easy to find a counterexample for that.)

So, we need a staying ahead measure that depends on the call being answered — instead of the total waiting time accumulated for the callers whose calls have been answered, consider the total waiting time due to callers waiting on this call. In the end it amounts to the same thing — the overall sum is the sum of the amount of time person $i$ waits on caller $j$ for all pairs $(i,j)$, so it doesn't matter if we sum over $i$ first or $j$ first.

> After $k$ calls have been added to the ordering, there aren't any repeated calls and the total waiting time due to those calls in the algorithm's solution is no bigger than the total waiting time due to those calls in an optimal solution.

There is a subtlety hidden here that would be good to clarify — the "after $k$ calls have been added to the ordering" refers to both orderings, the algorithm's and an optimal, so we are comparing the first $k$ calls in the algorithm's solution and the first $k$ calls in an 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.

For $k = 0$ (before the first iteration): with no calls answered, there aren't any repeats and no waiting time due to those calls in either solution.

For $k = 1$ (after the first iteration / before the second iteration): only one call has been answered, so there aren't any repeats. Let $A_1$ be the call the algorithm chose as the first call answered, and $O_1$ be the first call in an optimal solution. Every other caller has to wait on that call, so the total waiting time due to the algorithm's choice is $t_{A_1}(n-1)$ and the total waiting time due to the optimal's choice is $t_{O_1}(n-1)$. Since the algorithm chose the shortest call, $t_{A_1} \leq t_{O_1}$ and thus $t_{A_1}(n-1) \leq t_{O_1}(n-1)$ — the total waiting time due to the algorithm's choice is no bigger than the total waiting time due to the optimal's choice.

***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.

Assume the loop invariant is true for $k$: After $k$ calls have been added to the ordering, there aren't any repeated calls and the total waiting time due to those calls in the algorithm's solution is no bigger than the total waiting time due to those calls in an optimal solution.

We need to show that the invariant is maintained with the addition of call $k+1$. What happens when call $k+1$ is answered?

The algorithm picks the shortest remaining call, so it isn't already in the ordering. No repeats are introduced because of that call.

Proof by contradiction is a common tactic for the optimality part.

Assume that call $k+1$ is where the algorithm falls behind — the total waiting time due to the first $k+1$ calls in the optimal solution is less that the total waiting time due to the first $k+1$ calls in the algorithm's solution.

Let $A_i$ be the algorithm's $i$th call and $O_i$ be the optimal's $i$th call. Also let $T_{A_j}$ be the total waiting time due to the first $j$ calls in the algorithm's solution, and $T_{O_j}$ be the total waiting time due to the first $j$ calls in an optimal solution.

The invariant gives us that $T_{A_k} \leq T_{O_k}$.

Assuming that this is the step where the algorithm falls behind means that $T_{A_{k+1}} > T_{O_{k+1}}$.

$$T_{A_{k+1}} = T_{A_k} + t_{A_{k+1}}(n-k-1)$$

because $n-k-1$ callers wait on call $k+1$. Similarly,

$$T_{O_{k+1}} = T_{O_k} + t_{O_{k+1}}(n-k-1)$$

The combination of $T_{A_k} \leq T_{O_k}$ and $T_{A_{k+1}} > T_{O_{k+1}}$ means that $t_{A_{k+1}}(n-k-1) > t_{O_{k+1}}(n-k-1)$ and thus $t_{A_{k+1}} > t_{O_{k+1}}$.

To simplify the discussion for a moment, let's assume that no two calls have the same length. Since the algorithm always picks the shortest remaining call, there are exactly $k$ calls shorter than call $A_{k+1}$. The optimal's call $k+1$ is one of those calls because $t_{A_{k+1}} > t_{O_{k+1}}$, but that means one of the optimal's first $k$ calls must be longer than $A_{k+1}$ (and thus also $O_{k+1}$) — the optimal can't have $k+1$ calls all shorter than $A_{k+1}$.

Let's call that longer call $O_j$ ($j \leq k$). The total waiting time due to call $O_j$ is $t_{O_j}(n-j)$ and the total waiting time due to call $O_{k+1}$ is $t_{O_{k+1}}(n-k-1)$. Since $j \leq k$, $n-j > n-k-1$ and

swapping calls $O_{k+1}$ and $O_j$ in the optimal's ordering would result in a lower total waiting time:

$$\begin{aligned} t_{O_{k+1}}(n-j) + t_{O_j}(n-k-1) &\overset{?}{<} t_{O_j}(n-j) + t_{O_{k+1}}(n-k-1) \\ t_{O_{k+1}}(n-j) - t_{O_{k+1}}(n-k-1) &\overset{?}{<} t_{O_j}(n-j) - t_{O_j}(n-k-1) \\ t_{O_{k+1}}(k-j+1) &\overset{?}{<} t_{O_j}(k-j+1) \end{aligned}$$

which is true because $t_{O_j} > t_{A_{k+1}} > t_{O_{k+1}}$.

This means that the optimal's ordering (with $O_j$ before $O_{k+1}$) wasn't optimal, so the initial (and only) assumption that $T_{A_{k+1}} > T_{O_{k+1}}$ must be false.

But what if calls can have the same length? Then there are at most $k$ calls shorter than $A_{k+1}$; if there were more, one of them would have been picked instead of $A_{k+1}$. (There could be fewer than $k$ because several calls might be the same length as $A_{k+1}$.) That means that there is at least one call $O_j$ ($j \leq k$) in the optimal's solution where $t_{O_j} \geq t_{A_{k+1}}$. Because $t_{A_{k+1}} > t_{O_{k+1}}$ and $t_{O_j} > t_{O_{k+1}}$, the same reasoning as above means that swapping $O_j$ and $O_{k+1}$ would reduce the total waiting time in the optimal solution, so the optimal solution isn't optimal.

Since the assumption that this is the step where the algorithm falls behind led to the conclusion that the optimal solution isn't optimal, it must not be the case that the algorithm falls behind and the loop invariant holds.

***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.

The loop invariant gives us that after $k$ calls have been added to the ordering, there aren't any repeated calls and the total waiting time due to those calls in the algorithm's solution is no bigger than the total waiting time due to those calls in an optimal solution.

The loop exits when $n$ calls have been added to the ordering. Since there have been $n$ repetitions and no calls have been repeated, every call is in the ordering. The loop invariant gives the rest: with all $n$ calls in the ordering, there aren't any repeats and the total waiting time in the algorithm's solution is no bigger than the total waiting time in an optimal solution. Since the algorithm's solution can't actually be better than an optimal solution, it must be equal to an optimal solution and thus optimal itself.

***Implementation.*** Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

The ordering is to take the calls in order from shortest to longest, so the algorithm can be implemented by simply sorting the list by time.

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

$O(n \log n)$ to sort by time.