7.3 Callers On Hold

Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

Establish the problem.

- *Specifications.*

  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.*

Identify avenues of attack.

- *Targets.*
- *Approach.* Series of choices.

  (This was dictated.)

- *Paradigms and patterns.*

  Paradigm: greedy.

  (Again, this was dictated.)

  Identify the flavor, if applicable. Consider how this problem fits into each applicable pattern.

  This is an ordering problem.

  Produce output: repeatedly determine the next call to answer.

  Process input: figure out where in the ordering to answer the current call.

- *The series of choices.*

  Produce output seems much easier here.

  The next call to answer.

- *Greedy choices and counterexamples.*

  To identify the possible greedy choices, identify what information there is to work with.

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

  The only plausible options for picking the next call to answer is to pick the shortest remaining call.

  Picking the longest remaining call doesn't make sense because everyone else will have to wait for that call, and answering a shorter call instead reduces all those wait times. And that argument illustrates why picking any call doesn't work — the order does matter.

Define the algorithm.

- *Main steps.*

  ```
  repeatedly
    answer the shortest call remaining
  ```

- *Exit condition.*

  When all the calls have been answered.

- *Setup.*

  Nothing to do.

- *Wrapup.*

  Nothing to do — the ordering is the desired result.

- *Special cases.*

  If two calls are the same length, either one can be answered first.

- *Algorithm.*

  n people are on hold, waiting for a single tech support operator. 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

  ```
  repeatedly
    answer the shortest call remaining  (pick any such call if there's a tie)
  ```

Show termination and correctness.

- *Termination.*

  - *Measure of progress.*
    The number of calls answered.
  - *Making progress.*
    Another call is answered in each iteration.
  - *The end is reached.*
    Since calls are not answered twice, eventually all will have been answered.

- *Correctness.*

  - *Loop invariant.*

    The general pattern is

    ```
    After k iterations, ...
    ```

    The loop invariant needs to address both legality and optimality. For optimality, try a staying ahead argument. Since the task is to find an ordering, we are not free to consider the optimal's elements in any order — the apples-to-apples task is compare the first k calls in the algorithm's ordering to the first k calls in the optimal's ordering.

    But what should the staying ahead argument be? It's tempting to state something about the total waiting time so far with the k calls that have been answered, but keep in mind that to establish and maintain the invariant we need to make an argument involving the current call picked — but the total waiting time for the first k callers has nothing to do with the

> kth call. Instead we need to consider the impact of the kth call on the waiting times on the callers after caller k.

After k calls have been answered, the total waiting time due to the k calls in the algorithm's solution is no bigger than the total waiting time due to the first k callers in the optimal solution.

> "No bigger" because the goal is to minimize the total waiting time, so smaller is ahead.

– *Establish the loop invariant.*

> Show for $k = 1$ because $k = 0$ is too trivial — it doesn't even involve the greedy choice.

The total waiting time due to the algorithm's first call is $(n-1)t_{A_1}$ and the total waiting time due to the algorithm's first call is $(n-1)t_{O_1}$. Since the algorithm picks the shortest call, $t_{A_1} \leq t_{O_1}$ and thus $(n-1)t_{A_1} \leq (n-1)t_{O_1}$.

– *Maintain the loop invariant.*

Assume that after k calls have been answered, the total waiting time due to the k calls in the algorithm's solution is no bigger than the total waiting time due to the first k callers in the optimal solution.

Show that after $k+1$ calls have been answered, the total waiting time due to the $k+1$ calls in the algorithm's solution is no bigger than the total waiting time due to the first $k+1$ callers in the optimal solution.

Assume this is the step where things go wrong: after $k+1$ calls have been answered, the total waiting time due to the $k+1$ calls in the algorithm's solution is longer than the total waiting time due to the first $k+1$ callers in the optimal solution.

Let A_i be the algorithm's ith call and O_i be the optimal's ith 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 $T_{A_k} \leq T_{O_k}$. If the algorithm falls behind here — $T_{A_{k+1}} > T_{O_{k+1}}$ — it must be the case that the algorithm's $(k+1)$st call is longer than the optimal's $(k+1)$st call — $t_{A_{k+1}} > t_{O_{k+1}}$ — because the same number of people have to wait for call $k+1$ in both orderings.

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}.

> The idea of the contradiction is to show that there's must then be something shorter later in the optimal's solution that could be swapped with the longer call, reducing the total waiting time in the optimal solution and meaning that it isn't optimal.

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.*

> Show that the setup + main steps + wrapup yields the final answer. Since there aren't any setup or wrapup steps, show that the loop invariant plus the exit condition results in the final answer.

The loop invariant gives us that after k calls have been answered, the total waiting time due to the k calls in the algorithm's solution is no bigger than the total waiting time due to the first k callers in the optimal solution. The exit condition is that all n calls have been answered — if the algorithm's solution is still no worse than the optimal after all n calls, it must be optimal.

Determine efficiency.

- *Implementation.*

  Sort the calls by increasing length — that's the ordering desired.

- *Time and space.*

  $O(n \log n)$ to sort the calls.

- *Room for improvement.*

  > The algorithm is just sorting — hard to be $O(n \log n)$ for that!