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.

Print Job Scheduling

Print Services has n jobs to complete, and each job i has a length t_i and a deadline d_i. A feasible schedule is one in which all of the jobs get done before their respective deadlines. Find a feasible schedule if one exists.

(Note: While this isn't an optimization problem as such, there is an optimization version which, if solved, addresses the original problem. Solve the problem that way: Develop a greedy algorithm to find a schedule which minimizes the amount by which the most-late job misses its deadline. Then explain how finding such a schedule allows you to solve the original problem.)

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.

The original problem:

Task: Find a feasible schedule or determine that none exists.

Input: length t_i and deadline d_i for each job

Output: a feasible schedule, or that no such schedule exists

Since this problem will be solved by finding the schedule which minimizes the amount by which the most-late job misses its deadline, we'll apply the process to that version instead:

Task: Find the schedule which minimizes the amount by which the most-late job misses its deadline.

Input: length t_i and deadline d_i for each job

Output: a schedule (ordering of jobs)

Legal solution: an ordering (every job included, no job repeated)

Optimization goal: minimize the amount of time by which the most-late job misses its deadline

Targets.

No constraints were given.

The output is an ordering, and there are $O(n!)$ possible orderings of jobs.

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 task.

Process input: for each job, determine where it goes in the ordering

Produce output: repeatedly determine the next job in the ordering

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.

We have the length and deadline for each job, so possible greedy choices include: shortest job first (get more jobs done sooner), earliest deadline first (get the most urgent jobs done first), and smallest slack time $d_i - t_i$ (get the jobs with the least leeway on their deadlines done first).

The opposite choices (longest job first, latest deadline first, and largest slack time) are also possible, but there's not a plausible reason why any of them would be advantageous.

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

To determine which greedy choice to pursue, attempt to come up with counterexamples —

For shortest job first, we're looking for a case where the longer job could be completed by its deadline if it was done first but not if it is done second. Let job A have length 10 and deadline 10 and job B have length 2 and deadline 12. Shortest job first schedules B and then A; B finishes before its deadline but A misses by 2. If A was scheduled first, both jobs would meet their deadlines.

For smallest slack first, we're looking for a case where the smaller slack involves a later deadline so that doing that job first causes another job to miss its deadline. For example, job A has length 10 and deadline 15 and job B has length 3 and deadline 12. $d_i - t_i$ is smallest for A so it is done first, but that means that B doesn't finish until time 10+3, missing its deadline. On the other hand, if B is done first then both jobs meet their deadlines (B finishes at time 3 and A finishes at time 13).

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

For ordering problems, it can be easier to think about the produce output approach because it focuses attention on the choice — how the choose the next job — and because it keeps the partial solution intact — new things are only appended.

Produce output: Repeatedly add the job with the next earliest deadline to the end of the schedule.

Exit condition. Identify when the loop ends.

The loop ends when all of the jobs have been added to the schedule.

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

Sort the jobs by earliest deadline.

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

To find the schedule that minimizes the amount by which the most-late job misses its deadline, nothing else is needed. (The order is the schedule.) For the original task, determine how much the most-late job misses its deadline in the schedule — the schedule is feasible if that value is 0 and not otherwise.

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

Multiple jobs may have the same deadline. In that case, the order doesn't matter — the second of the two jobs will finish at the same time regardless of which is done first, as will everything after the second job.

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

`schedule(jobs)` — find a feasible schedule or determine that none exists

Input: length t_i and deadline d_i for each job

Output: a schedule (ordering of jobs)

Legal solution: a feasible ordering (every job included, no job repeated, no job misses its deadline), or that no such schedule exists

```
      repeat
        add the job with the next earliest deadline to the end of the schedule

      if the amount by which the most-late job misses its deadline is 0
        return the schedule as a feasible solution
     otherwise report no feasible solution
```

Termination. Show that the loop eventually terminates.

Measure of progress. The number of jobs which have been scheduled.

Making progress. In each iteration, another job is added to the schedule.

Reaching the end. With a new job scheduled in each iteration, eventually all will have been scheduled.

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.)

We start with a standard staying ahead pattern for the correctness part of the invariant: the most late job in the algorithm's schedule-so-far is no later than the most late job in optimal's schedule-so-far. A question is what constitutes the optimal's schedule-so-far — the first k jobs in it or the algorithm's first k jobs. However, if we try to compare the first k jobs in both, we can't show that the loop invariant holds for $k = 1$ — we don't know anything about what is first in the optimal schedule, so it could be a job that makes its deadline while the algorithm's first job (the earliest deadline) misses its.

Then, when trying to make the argument for maintaining the invariant, it turns out that what is really needed in this case is something of a combination of the two approaches — to compare the first k jobs in the algorithm's schedule to a "compressed optimal" schedule consisting of only those k jobs but in the order of the optimal schedule.

After k jobs have been scheduled, no job occurs in the schedule more than once (legality) and the most late job is no later than the most late of those same k jobs scheduled in the order in which they appear in the optimal schedule (correctness).

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 jobs have been scheduled, so nothing is late — the amount that the most-late job misses its deadline by is 0 in both algorithm and optimal solutions.
>
> *$k = 1$.* There is only one job so far, so it is in the same order in the algorithm's schedule and the optimal schedule and thus can't be any later in the algorithm's schedule than if scheduled in the order of the optimal schedule.

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.

> *No repeated jobs.* The loop invariant gives us that no job is repeated amongst the first k programs, so if there's a repeat after $k + 1$ jobs, it is because job $k + 1$ was already in the schedule. But each job is only added to the schedule once, so that's impossible.
>
> *The most-late job is no more late than the most late of the same jobs scheduled in the same order as in the optimal schedule.* The loop invariant gives us that after k jobs have been scheduled, the most-late job is no more late than the most late of the same k jobs scheduled in the same order as in the optimal schedule. Assume that this is the point when things go wrong — after $k + 1$ jobs, the most-late job is now later in the algorithm's schedule than with the optimal's ordering. What can we conclude?
>
> - Job $k + 1$ must miss its deadline in the algorithm's solution. If not, the max lateness of the algorithm's first $k + 1$ jobs would be the same as the algorithm's first k jobs and, since max lateness cannot decrease with the addition of more jobs, the algorithm's max lateness could not now exceed the lateness with the optimal's ordering.
> - Furthermore, job $k + 1$ must miss its deadline in the algorithm's schedule by more than it misses its deadline with the optimal's ordering by a similar argument.
> - Job $k + 1$ must not be last in the optimal's ordering — it must start earlier than it does in the algorithm's ordering in order to not miss its deadline by as much, which means there must be at least one job that comes before job $k + 1$ in the algorithm's ordering but after $k + 1$ in the optimal's ordering.
>
> Let j be the job immediately after job $k + 1$ in the optimal's ordering. (This exists because of the last observation above.) $d_j \leq d_{k+1}$ because the algorithm scheduled j before $k + 1$ and the algorithm picks jobs in increasing order of deadline. Consider the two cases:
>
> *$d_j = d_{k+1}$* Swapping j and $k + 1$ in the optimal's ordering would not increase the lateness of the most late job in the optimal's ordering. Since the deadline is the same for both jobs, the second one to finish is the one that exceeds the deadline by the most, and the end time of the second job to finish is the same regardless of whether j or $k + 1$ is done first.
>
> *$d_j < d_{k+1}$* Swapping j and $k + 1$ in the optimal's ordering would not increase the lateness of the most late job in the optimal's ordering. Doing j first means it ends earlier, decreasing the amount by which it exceeds its deadline (if it does), and while doing $k + 1$ second means it finishes later than it did, it finishes at the same time that j did before the swap and its deadline is later so it will not exceed its deadline by as much as j exceeded its own deadline.
>
> So, the algorithm falling behind in this step means that job $k + 1$ wasn't the last-scheduled in the optimal's ordering, but the order in which the algorithm picks jobs means that job $k + 1$

could be swapped with the job after it in the optimal's ordering, resulting in a new ordering where the most late job is no later than before the swap — so this swap didn't make the optimal ordering non-optimal. The process can be repeated until job $k + 1$ is at the end of the optimal's ordering, which means that it can't miss its deadline by more in the algorithm's ordering than in the optimal's (because the same k jobs precede job $k + 1$ in both orderings) and thus the assumption that this is where the algorithm falls behind is false.

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 jobs are in the schedule; plugging $k = n$ into the loop invariant gives us that after n jobs have been scheduled, no job is repeated and the most late job is no later than the most late of those same n jobs in order in which they appear in the optimal schedule. Since n jobs are all that are in the optimal schedule, this means that the most late job in the algorithm's schedule is no later than the most late job in the optimal schedule. And since the algorithm can't do better than the optimal (if it did, the optimal solution wouldn't be optimal), it must be that the most-late job in the algorithm's schedule is just as late as in the optimal schedule i.e. the algorithm produces the correct solution.

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.

Repeatedly finding the job with the next earliest deadline can be handled efficiently by sorting the jobs.

Time and space. Assess the running time and space requirements of the algorithm, providing sufficient implementation details (data structures, etc) to justify those answers.

It takes $O(n \log n)$ time to sort the jobs, and that is all that is required for the schedule that minimizes the amount by which the most-late job misses its deadline. Determining if the schedule is feasible can be done in $O(n)$ time by going through the schedule and comparing the finishing time for each job to its deadline.