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.

The Time-O Problem

Orienteering is the sport of cross-country navigation - competitors must navigate to a series of checkpoints (called controls) using a map and compass. An orange-and-white flag marks the location in the terrain. Controls typically must be visited in a particular order, and the goal is to navigate to a certain sequence of controls as quickly as possible.

In contrast to point-to-point events, controls in a score-o event have associated point values and may be visited in any order. However, there is a time limit - the task is to select which controls to visit (and in what order) so as to maximize your score within the specified time limit. A penalty is assessed for each minute (or part of a minute) overtime.

In the novelty time-o variant, there is an additional constraint that each control is only available during a particular time window. Visiting a control outside of its time window incurs no penalty, but also gains no points.

The following assumptions will be made for this problem:

- The start and finish are at the same location.
- The penalty is a fixed number of points per minute, specified as part of the problem input.

A few other notes:

- There are no restrictions on the direction of travel between controls, but the speed of travel in one direction may be very different from the other direction. (e.g. going in the uphill direction vs the downhill direction)
- There is no penalty for visiting a particular control more than once (or outside its time window), but only one visit (within the time window) counts for scoring.
- Travelling between controls is assumed to always occur at the runner's maximum pace, but it is allowed to stop and wait for any length of time at a control. (It might be advantageous to arrive at a control early and wait for its time window to open.)

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: Given a list of controls with their point values and available time windows, the overall time limit and overtime penalty (points per minute), and the time it takes to travel between each pair of controls (including start and finish), find which controls to visit and in what order so as to maximize your score.
>
> Input: n controls, each with point value p_i and time window $[s_i,f_i]$; the overall time limit T; overtime penalty P points per minute; time t_{ij} to travel between each pair of controls (i,j)
>
> Output: list of controls visited, in order, with the time of each visit and the total score
>
> Legal solution: the controls visited start with the start control and end with the finish control
>
> Optimization goal: maximum score

Size. What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

The number of choices left to make depends on the solution length (the number of controls visited); it is at most n, the number of controls (not counting the start and finish).

The smallest problem in terms of the input size is $n = 0$ — no controls. The smallest problem in terms of the number of remaining decisions is 0 — at the finish.

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

No constraints on time or space are given in the problem.

Since it is legal to visit controls more than once and to exceed the time limit, there are infinitely-long legal solutions. But it is unlikely that visiting controls more than once will be advantageous, so there are $n!$ possible orderings.

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.

Recursive backtracking is exponential at best, but we still want to minimize the work done per subproblem — ideally O(1).

Approaches.

This is a find-the-best-solution task.

This is an ordering task. (While the controls visited may be a subset of the full set of controls, this isn't a subset problem because the order in which the controls are visited matters.)

Process input: for each control, put it where it belongs in the ordering (or decide that it isn't included).

Produce output: repeatedly determine the next control to visit

While more of the input will tend to have a lower branching factor, especially if most routes don't clear the course or come close to clearing and so have fewer than n controls, adding a control into an existing ordering has a complex effect on that route's score because subsequent visits will be pushed later and may now miss a control's window entirely. That seems expensive to compute, never mind determining whether there might be a correctness issue, so more of the output is the better choice here.

Generalize / define subproblems. Friends get smaller versions of the original problem. Define the task, input, and output for the subproblems.

Partial solution. Identify what constitutes a solution-so-far. This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).

The controls visited so far, in order, with the time of each visit and the total score.

Alternatives. Identify the choice being made and what the (legal) alternatives are for that choice. (Alternatives that result in an illegal solution aren't considered farther.)

With process output, the choice is which control to visit next.

It is possible to go directly from any control to any other control, so any control (including the finish) could be the next control visited. The only thing not eligible as the next control is the start.

Subproblem. The "rest of the problem". Define the generalized problem, its input, and its output. The input can include the partial solution so far, in which case the output would be a complete solution, or the input can only denote the subproblem, in which case the output is only the solution for the subproblem.

> Task: timeo(*current*,*time*,*punched*) — find a route (which controls to visit and in which order) from *current* to the finish which maximizes the score given controls already *punched*
>
> Input: the current location (*current*) and time (*time*); *punched* is the controls which have already earned points
>
> Output: list of the controls visited, in order, with the time of each visit and the total score earned

Global constant parameters (such as the collection of all of the controls with their point values and open windows, and the time limit and penalty) don't need to be included in the parameters — the goal here is to define what parameterizes the subproblem and distinguishes one subproblem from another. In an actual implementation these will need to be passed, but the goal of developing the algorithm is working out the ideas — and what matters for that is what information there is, not how exactly it is represented or accessed.

Main case. This takes the form of: for each legal alternative for the current decision, a friend solves the remainder of the problem given the solution so far plus that choice. We return one of those solutions, the best of those solutions, or all of the those solutions depending on the structural variation.

At each control there are potentially two choices, move on immediately after punching or waiting for the control to become available before punching and moving on.

```
for each control c (including the finish)

  // arrival time at c and the points earned punching it then
  arrival_c ← time + t_current,c
  points_c ← points(c,arrival_c,punched)

  // find the rest of the route to the finish from c, recording c as punched if
  // points were earned
  if points_c > 0
    (route_c,score_c) ← timeo(c,arrival_c,punched ∪ c)
  otherwise
    (route_c,score_c) ← timeo(c,arrival_c,punched)

  if the arrival time is before c's window opens at time s_c    // wait for it to open
    points_c' ← points(c,s_c,punched)
    // find the rest of the route to the finish after waiting, recording c as
    // punched if points were earned
    if points_c' > 0
      (route_c',score_c') ← timeo(c,s_c,punched ∪ c)
    otherwise
      (route_c',score_c') ← timeo(c,s_c,punched)

return (c followed by route_c,score_c+points_c) for the c which maximizes
 score_c+points_c) (including waiting until the time window opens)
```

Base case(s). A base case occurs when there are no more choices to make.

> A complete route is when the finish has been reached — `timeo(finish,time)`. In that case, return `( (), -penalty )` where `penalty` is 0 if $time \leq T$ and $P * (time - T)$ otherwise.

A wrinkle here is that infinite (or, perhaps better phrasing, arbitrarily long) solutions are legal because controls can be revisited and the time limit can be exceeded. Showing termination, however, requires that that a base case can be reached.

In addition, a route is "functionally complete" — there are no more points to be gained and no reason not to return to the finish — when all of the time windows have closed. In that case, return `((finish),-penalty)` where `penalty` is 0 if $time+t_{current,finish} \leq T$ and $P*(time+t_{current,finish}-T)$.

Top level. The top level puts the context around the recursion.

Initial subproblem. `timeo(start,0)` — we begin at the start with no time elapsed

Setup. Nothing needed.

Wrapup. Return the route and score.

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

Small inputs, such as $n = 0$ or $n = 1$, can be special cases. While they can be treated separately (both are easy to detect and the correct solution, especially for $n = 0$, is fairly simple), there's no need to — the algorithm as written works fine. (Check this for yourself.)

Non-unique (tie) values may require special handling when choices are being made based on those values, but there are no such choices here — the time window's opening time is used only to determine to determine if that control is a possibility for the next control, not to choose between two controls to decide which is next.

None.

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

`timeo(controls,T,P,times)` — find a route (which controls to visit and in which order) from the start to the finish which maximizes the score

Input: n controls, each with point value p_i and time window $[s_i,f_i]$; the overall time limit T; overtime penalty P points per minute; time t_{ij} to travel between each pair of controls (i,j)

Output: list of controls visited, in order, with the time of each visit and the total score

Legal solution: the controls visited start with the start control and end with the finish control

```
timeo(start,0,{})
```

`timeo(`*`current`*`,`*`time`*`,`*`punched`*`)` — find a route (which controls to visit and in which order) from *current* to the finish which maximizes the score

Input: the current location (*current*) and time (*time*)

Output: list of the controls visited, in order, with the time of each visit and the total score earned

```
if current is the finish
  return ((),0) if time ≤ T and
         ((),−P∗(time−T)) otherwise

else if all of the time windows have closed
  return ((finish),0) if time + t_{current,finish} ≤ T and
         ((finish),−P∗(time + t_{current,finish} − T)) otherwise

otherwise

 for each control c (including the finish)

   // arrival time at c and the points earned punching it then
   arrival_c ← time + t_{current,c}
   points_c ← points(c,arrival_c,punched)

   // find the rest of the route to the finish from c, recording c as punched if
   // points were earned
   if points_c > 0
     (route_c,score_c) ← timeo(c,arrival_c,punched ∪ c)
   otherwise
     (route_c,score_c) ← timeo(c,arrival_c,punched)

   if the arrival time is before c's window opens at time s_c    // wait for it to open
     points_{c'} ← points(c,s_c,punched)
     // find the rest of the route to the finish after waiting, recording c as
     // punched if points were earned
     if points_{c'} > 0
       (route_{c'},score_{c'}) ← timeo(c,s_c,punched ∪ c)
     otherwise
       (route_{c'},score_{c'}) ← timeo(c,s_c,punched)

  return (c followed by route_c,score_c+points_c) for the c which maximizes
   score_c+points_c) (including waiting until the time window opens)
```

Termination. Show that the recursion ends. For making progress, explain why each subproblem is smaller. For reaching the end, show that a base case is always reached.

Making progress. Assuming $t_{ij} > 0$ (a valid assumption, since otherwise multiple controls would be at the same location), time passes with each step.

Reaching the end. Assuming finite time windows (again a reasonable assumption, as there's no reason for the course designer to have time windows open beyond the course time limit), time passing means that eventually all of the time windows will close.

Establish base case(s). Explain why the solution is correct for each base case.

When the finish is reached, the route is a complete route — the current control, which is the most recent one finished, is the finish. The route is also a legal route because any ordering of controls that ends with the finish is legal. No points are gained at the finish, but a penalty may be assessed, so that is taken into account.

When the time windows run out, the route must be completed by heading to the finish. That additional stop on the course and any penalties based on the time of arrival at the finish are

accounted for.

Show the main case. Explain why all possible alternatives for the next choice are covered and that the right partial solution is passed to each friend. Then, assuming that the friends return the correct results for their subproblem, explain why the correct answer is produced from those results.

Any control (including the finish) is legal as a next control, and there are always two options: punch and move on right away, or wait before punching and moving on. There are an infinite number of possibilities for the waiting time, but the only one that makes sense is to wait just long enough for a time window to open.

Final answer. Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

The race starts at the start with no time elapsed and no controls punched i.e. `timeo(start,0,{})`. This returns the whole route and the total score (including penalties), which is exactly what is needed for the final result.

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.

Determining if a control has earned points and adding a control to *punched* is O(1) if *punched* is a hashtable-backed set. (Removing, needed to avoid copying sets, is also O(1).)

Prepending a control to a route is O(1) for a linked list.

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

The branching factor n. The longest path length depends on how many controls can be visited before all of the time windows expire, but pruning visits where no points are earned (likely to be safe) means that it is n, yielding $O(n^n)$ subproblems.

The total running time includes the work done in each subproblem. With the implementations identified above, the work per subproblem is O(1) so the total running time is $O(n^n)$.

This is terrible, so the next step would be to consider pruning. While it is legal to visit controls more than once or to visit outside the open window, there is nothing to be gained if no points are earned — there is information about the time to get from any control to any other, so presumably if the best route involved going by a third control, that would already be included in the time. This means that possible next controls can be pruned if we'd arrive after the open window closes, and that if we arrive before the window opens, we should always wait until the window opens. That also means that the first visit to a control will always be the one that earns points, so any previously-visited control can be pruned.

Thus, the main case can be updated so that only unvisited controls where we arrive before the window closes (plus the finish) are considered, and that the arrival time is taken as the later of the actual arrival time and the start of the control's window.

What about other pruning, such as taking into account the time limit? It is legal to exceed it — while that means losing points, it is possible that it could be advantageous to be late if that means visiting an extra control (or several). Since the goal at this stage is to develop the basic algorithm, we'll leave more complex pruning for later.