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.

Time-O

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.)

Establish the problem.

Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *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, identify and 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, the difference in timestamps between successive controls is no less than the time needed to travel directly between those controls, and the total score is correctly computed (at most one visit per control and only visits within each control's time window are counted, overtime penalty is assessed)

Optimization goal: maximum score

- *Examples.*

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

Identify avenues of attack.

> Determine the algorithmic approach(es) to try.

- *Targets.*

  > Backtracking algorithms are fundamentally exponential.

- *Paradigms and patterns.*

  Paradigm: backtracking.

  > (This is dictated.)

  > Consider the backtracking patterns.

  This is an ordering task — the order the controls are visited matters — but duplicates and omissions are allowed (controls can be visited more than once, or not at all).

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

  Produce output: repeatedly find the next control to visit

  > Process input is often more awkward for ordering tasks, and is not appropriate here when duplicates and omissions are allowed in the ordering.

- *The series of choices.*

  > Identify the series of choices. Take into account efficiency considerations when choosing between alternatives.

  The next control to visit.

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*

  > The size of the problem is the number of choices left to make. The smallest problem size is 0 (a complete solution).

  The number of choices left to make = the number of controls left to visit. This can be arbitrarily many, since controls can be visited more than once and the time limit can be exceeded, but in practice is not likely to be much more than n, the number of controls, because visiting a control more than once doesn't increase the score and would only be advantageous if it is faster to go from control A to control B to control C than to go straight from A to C.

  $n = 0$ occurs when the finish has been reached.

- *Generalize / define subproblems.*

  - *Partial solution.*

> 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 output is the list of controls visited, in order, with the time of each visit and the total score. A partial solution is the sequence of controls that have been visited so far with the time of each visit and the score accumulated by those visits.

– *Alternatives.*

> What are the legal alternatives for the next choice?

Any control, other than the current one, can be visited next. The finish can also be visited next.

It is also possible to wait at a control. This could be handled by including the current control as one of the choices for the next control.

– *Subproblem.*

> Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified. Take into account efficiency considerations when defining the input and output.

The subproblem's task is to solve the rest of the problem in light of the choices made so far — complete the route from the current control to the finish so as to maximize the total score for the whole route, given the controls already visited and the current time.

Task: `timeo`(*route*,*timestamps*,*current*,*time*,*score*) — complete the route from the current control to the finish so as to maximize the total score given the controls already visited and the current time

Input: the sequence of controls visited so far (*route*), when each was visited (*timestamps*), and the current control (*current*), time (*time*), and score (*score*)

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

> The current score can be computed from the route and timestamps, but including it directly as part of the input saves computation time.
>
> 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.

- *Base case(s).*

> A complete solution has been found — address what to do with it.

If the current control is the finish, the solution is complete — return *route*, *timestamps*, and *score.*

- *Main case.*

> Address how to solve a typical large problem instance (one that is not a base case).

> The pattern for "find the best solution" —
>
> ```
> best result so far ← no solution
>
> for each legal alternative for the current choice,
>   result = solve the subproblem resulting from choosing that alternative
>
>   if result is a solution and better than the best result so far,
>     best result so far ← result
> ```

```
return the best result so far
```

For each possible next control, add it to the route and solve the rest of the problem from there.

```
best result so far ← no solution

for each control $c$ (including the current control and the finish)

  // only wait at $c$ if it will be possible to punch it in the future
  if $c == current$,
    if $time \geq$ the start of $c$'s time window, continue
    otherwise $arrival \leftarrow$ the start of $c$'s time window
  otherwise $arrival \leftarrow time +$ travel time to $c$

  // add $c$ as the next control to visit
  append $c$ to $route$
  $timestamps[c] \leftarrow arrival$
  if $arrival$ is within $c$'s time window and $c$ has not previously been
    punched, $newscore \leftarrow score + points[c]$
  otherwise $newscore \leftarrow score$

  // solve the rest of the problem
  $result$ = timeo($route$,$timestamps$,$c$,$arrival$,$newscore$)

  // did we find a new better solution?
  if there is no best result so far or $result$'s score > best score so far
    best result so far ← $result$

  // undo $c$ as the next control to visit
  remove $c$ from $route$
  unset $timestamps[c]$
  // $score$ wasn't changed, so no need to undo

return the best result so far
```

- *Top level.*

  The top level puts the context around the recursion.

  - *Initial subproblem.*

    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    ```
    $route \leftarrow start$
    $timestamps[start] \leftarrow 0$
    timeo($route$,$timestamps$,$start$,0)
    ```

  - *Setup.*

    Whatever must happen before the initial subproblem is solved.

  - *Wrapup.*

    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

- *Special cases.*

> Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

> Empty or small inputs — in this case, referring to the number of controls — can be special cases. Duplicate (tie) values may require special handling when choices are being made based on those values.

Having no controls or only one control other than the start/finish requires no special handling. All possible next controls are considered for each choice so there is never a situation where there is a need to choose between tie values.

Making progress requires that the travel time between controls be > 0. This should be the case (controls are at distinct locations, so it will take at least a short time to get from one to the next), so it can be a precondition of the problem and not a case that needs handling.

- *Algorithm.*

> Assemble the algorithm from the previous steps and state it.
>
> There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

> This includes a revision to the main case that was revealed when attempting to show making progress.

`algorithm timeo(`*controls*`,`*p*`,`*w*`,`*T*`,`*P*`,`*t*`)` —

Task: find which controls to visit and in what order so as to maximize your score

Input: n controls (*controls*), each with point value p_i and time window $w_i = [s_i, f_i]$; 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

```
route ← start
timestamps[start] ← 0
timeo(route,timestamps,start,0)
```

`algorithm timeo(`*route*`,`*timestamps*`,`*current*`,`*time*`,`*score*`)` —

Task: complete the route from the current control to the finish so as to maximize the total score given the controls already visited and the current time

Input: the sequence of controls visited so far (*route*), when each was visited (*timestamps*), and the current control (*current*), time (*time*), and score (*score*)

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

```
if current is the finish,
  return route, timestamps, score as the solution

otherwise
  best result so far ← no solution

  for each control c (including the current control and the finish)

    // ensure making progress --- do not continue visiting controls if there
    // are no more points to be gained
    if $c$ is not the finish and $time \geq$ the end of the latest-closing
      time window, continue

    // only wait at $c$ if it will be possible to punch it in the future
    if $c == current$,
      if $time \geq$ the start of $c$'s time window, continue
      otherwise $arrival \leftarrow$ the start of $c$'s time window
    otherwise $arrival \leftarrow time +$ travel time to $c$

    // add $c$ as the next control to visit
    append $c$ to $route$
    $timestamps[c] \leftarrow arrival$
    if $arrival$ is within $c$'s time window and $c$ has not previously been
      punched, $newscore \leftarrow score + points[c]$
    otherwise $newscore \leftarrow score$

    // solve the rest of the problem
    $result$ = timeo($route$,$timestamps$,$c$,$arrival$,$newscore$)

    // did we find a new better solution?
    if there is no best result so far or $result$'s score $>$ best score so far
      best result so far $\leftarrow result$

    // undo $c$ as the next control to visit
    remove $c$ from $route$
    unset $timestamps[c]$
    // $score$ wasn't changed, so no need to undo

  return the best result so far
```

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*

  | Show that the recursion — and thus the algorithm — always terminates.

  – *Making progress.*

    Explain why what each of your friends get is a smaller instance of the problem: in each step, another choice is made.

    In each instance of the main case, another control is visited — but the base case is reaching the finish, and since controls can be visited more than once, there *isn't* a guarantee that we'll run out of other choices and eventually have to choose the finish as the next control.

    However, there is a point beyond which continuing to visit controls will no longer improve the result — once all of the time windows have closed, no more points can be earned. Incorporate that into the main case.

    Now consider the revised main case, from the statement of the whole algorithm above —

  In each instance of the main case, another control c is visited and the current time increases: if c is not the current control, the arrival time at c is later than the current time as long as the travel time between controls is > 0 (a precondition), and if c is the current control, it is only selected as the next control (with an arrival time equal to the start of c's time window) if the start of c's time window is $>$ the current time.

  – *The end is reached.*

> Explain why a base case is always reached: eventually all choices have been made.

If the next control chosen is the finish, the base case has been reached.

If not, increasing the current time means that eventually it will become later than latest window closing time and then the only possible next control is the finish.

- *Correctness.*

  > Show that the algorithm is correct.

  - *Establish the base case(s).*

    > Explain why the solution is correct for each base case: the right thing is done with a complete solution.

    Reaching the finish means a complete solution, so it is returned.

  - *Show the main case.*

    > Explain why the desired solution will be found (all possible legal alternatives for the next choice are either considered or are safe to skip) and showing that the correct partial solutions and subproblems are handed to the friends.

    Since it is possible to wait at a control and also to visit controls more than once or outside their time windows, any control (along with the finish) is valid as a next choice. There are two situations where a solution might be pruned: the next control c is the same as the current control and the current time is past the start of c's time window, and the current time is past the end of all time windows. In the first case, it is not possible to gain additional points from c by waiting — either it is currently in c's time window and we just punched it (so punching again later won't gain points), or it is past c's time window and there is no more opportunity to punch and gain points. Waiting at c might be advantageous for other controls, for example, waiting at c might mean arriving at a future control within its time window instead of before — but we could then just wait at the future control instead. In the second case, there are no more points to be gained by visiting additional controls so heading to the finish next will result in a score at least as high as any other continuation from the current control. Thus, while valid solutions might be pruned, no better solution will be pruned.

    The subproblems passed to the friends update the route (by appending the next control) and timestamps (by adding the visit time for that next control), and the current control, time, and score reflect arriving at and punching that next control.

  - *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. For the initial subproblem, establish that the correct input is provided – the (empty) solution-in-progress is legal and the rest of the problem is actually the whole problem.

  The initial subproblem is to complete the route from the current control (the start) to the finish so as to maximize the total score given the controls already visited (none) and the current time (0) — that's the whole task.

Determine efficiency.

> Evaluate the running time and space requirements of the algorithm.

- *Implementation.*

  > Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

- *Time and space.*

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

> For a recursive algorithm, write a recurrence relation for the running time. Use the table discussed in class to find a big-Oh for each recurrence relation.

- *Room for improvement.*

  > Algorithm design decisions — the series of choices pattern, the specific input and output for the subproblems — should have already been considered, but revisit them if not.