Find a legal solution. If the subproblem output is a complete solution:

```
for each legal alternative for the current choice,
  result = solve the subproblem resulting from choosing that alternative

  if result is a solution,
    return it

return no solution
```

If the subproblem output is just the subproblem solution, then `return result+alternative` the result.

Find the best solution. If the subproblem output is a complete 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
```

If the subproblem output is just the subproblem solution, then replace `result` with `result+alternative` when checking whether the current alternative results in a better solution and when updating the best result so far.

Find all solutions.

```
for each legal alternative for the current choice,
  solve the subproblem resulting from choosing that alternative
```

The base case handles the complete solutions (outputting them or adding them to a collection of solutions).

key elements of a backtracking formulation –

– series of choices, seek to limit the branching factor

– friends complete the current partial solution, in light of the choices made so far

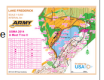
– base case is a complete (legal) solution

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 map shows a sample scenario; click on it to see a larger version. Controls are shown with purple circles and the start/finish is marked with a double purple circle. The three-digit label by each circle is the control code (so you can verify that you've arrived at the right flag) and the time window (in minutes) is shown in brackets. In this event, all controls were worth 1 point.



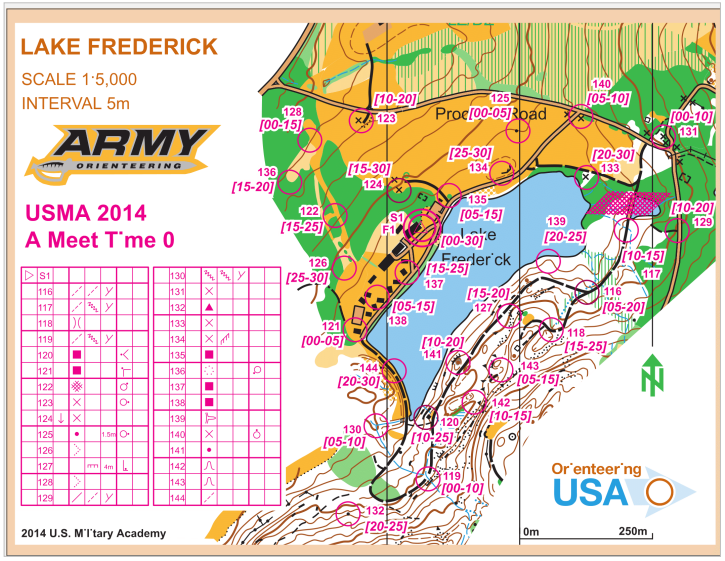
The following assumptions will be made for this problem:

- 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.)

The 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 the start and finish), find which controls to visit and in what order so as to maximize your score.



Running Time

How long does this take?

- DFS is O(n+m)
  - *n* = number of vertices, *m* = number of edges

How big is the state space graph?

- branching factor *b* – number of next choices
- longest path *h* – largest number of decisions needed to reach a base case

→ worst case $n = O(b^h)$, $m = O(b^{h+1})$

- if there are multiple paths to the same vertex, *n* can be much smaller – but without storing discovered vertices, repeat visits are handled the same as new visits (and storing discovered vertices takes exponential space)

This...is not good.

Key Points – Making Backtracking Practical

- recursive backtracking is generally not practical without additional effort
  - DFS is O(n+m) where n = $O(b^h)$
    - b = branching factor – number of next options for each choice
    - h = length of longest path – (maximum) number of choices made to get to a complete solution

Key Points – Making Backtracking Practical

- while reducing how much is explored is the dominating factor, it is also important to be efficient in what is done for each subproblem
  - determining whether or not to prune must be efficient
  - modify/restore rather than copying for generating subproblems and partial solutions
  - exploit clever representations

Generating New Partial Solutions and Subproblems

- making a choice typically means an incremental change to the current partial solution and subproblem
- generating the new by copying the old may be expensive
  - copying a collection takes time proportional to the size of the collection

Instead, it may be more efficient to modify the current partial solution and subproblem and then undo.

```
for each legal next choice
  add choice to partial solution and remove
    from subproblem
  result = solve(modified partial solution,
                 modified subproblem)
  remove choice from partial solution and add
    to subproblem
```


Generating New Subproblems

An appropriate framing of alternatives and a clever representation may also be effective.

- e.g. 0-1 knapsack
  - subproblem requires the set of items remaining to consider
    - if the items left to be considered are all at one end of an array, a single index k + the original collection of items is sufficient to define S for the subproblem
  - the series of choices and the alternatives
    - take or not take the next item (process input)
      - S is defined by k+1 for the subproblems
    - which item to take next (produce output)
      - combine with considering items in some order → S is defined by k+1, k+2, k+3, … for the subproblems