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.

Challenges

You are participating in a contest which consists of a series of challenges. The wrinkle is that completing a challenge takes time, and so doing challenge i will cause you to miss the next d_i challenges. (There's no way to go back to complete a missed challenge after the fact.) Given that you earn p_i points for completing challenge i (and that you complete any challenge you start), determine which challenges to do so as to maximize your score.

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: Determine which challenges to do to maximize your score, given that each challenge earns a certain number of points but also precludes completing some number of subsequent challenges.
>
> Input: for each challenge i, d_i (number of subsequent challenges missed if challenge i is completed) and p_i (points earned by completing challenge i)
>
> Output: which challenges to complete
>
> Legal solution: if challenge i is completed, no challenge $i + 1 < j < i + d_i + 1$ is completed
>
> Optimization goal: max 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 size of the problem is n, the number of challenges.
>
> The smallest problem is $n = 0$.

Approaches. Identify the particular flavor, if applicable, as well as the two patterns (process input and produce output) and what they look like for this problem.

> This is a "find a subset" problem — while the order of the challenges is important, that order is fixed. (It's not part of the task to determine which order to do the challenges in.)
>
> Process input: for each challenge, decide whether or not to complete it
>
> Produce output: repeatedly determine the next challenge to do

Process input has a lower branching factor, so that seems like a better approach from an efficiency standpoint — less work per subproblem.

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 challenges chosen so far.

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.)

> The choice is whether or not to do challenge k. It is only possible to do challenge k if it isn't blocked by the previously-chosen challenge.

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.

Also, with dynamic programming, we shift the output to be just the optimization value rather than the set of choices made.

> Original problem: Determine which challenges to do to maximize your score, given that each challenge earns a certain number of points but also precludes completing some number of subsequent challenges.
>
> Subproblem / generalized problem: Given challenges picked so far, find the maximum score possible from the remaining challenges, given that each challenge earns a certain number of points but also precludes completing some number of subsequent challenges.
>
> Input: challenges $k..n$ remaining; for each challenge i, d_i (number of subsequent challenges missed if challenge i is completed) and p_i (points earned by completing challenge i)
>
> Precondition: none of challenges $k..n$ are blocked by a previously-chosen challenge
>
> Output: max score
>
> Legal solution: if challenge i is completed, no challenge $i+1 < j < i+d_i+1$ is completed
>
> Optimization goal: max score

It is only legal to do a challenge if it isn't blocked by a previously-chosen one, but there isn't anything in the subproblem input to make such a check possible. However, observe that doing a challenge blocks some number of the subsequent challenges — that means that there's a clean division between the next challenges (blocked) and later challenges (available). As a result it is always possible to generate a subproblem where all challenges $k..n$ are available so we can solve this problem with a precondition instead of an additional input parameter.

Introducing notation, and focusing on the subproblem parameters that characterize the subproblem (rather than global information):

> `maxscore(k)` — the max score possible from challenges $k..n$
> Precondition: challenges $k..n$ are legal as the next challenge to complete

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.

```
maxscore(k) = max { maxscore(k + d_k + 1) + p_k,  // do challenge k
                    maxscore(k + 1) }             // don't do challenge k
```

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

A complete solution has been found when all of the challenges have been considered. In that case, there are no more challenges so that max score possible is 0.

```
maxscore(k) = 0        // k ≥ n
```

This is phrased as $k \geq n$ rather than just `maxscore(`n`)` because $k + d_k + 1$ could be greater than n if the d_k aren't carefully chosen.

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

> **Initial subproblem.** `maxscore(0)` — find the max score possible from challenges $0..n-1$
>
> **Setup.** Nothing to do.
>
> **Wrapup.** Reconstruct the list of challenges chosen.

The initial subproblem returns the maximum score, but the original problem wanted the challenges themselves.

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

> Nothing needed.

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.** k (the current challenge) increases by at least 1 in each step.
>
> **Reaching the end.** The base case is when $k \geq n$, so k eventually gets there.

Correctness. Explain why the base case answer is correct. Explain why the main case covers all of the legal next possibilities, the right parameters are passed to the subproblems, and answer is correct assuming correct subproblem answers. Explain why the parameter values for the initial subproblem are correct.

> **Establish the base case(s).** When there is a complete solution, there are no challenges left to choose and no more points to be earned.
>
> **Show the main case.** Choose challenge k or don't choose challenge k are the only possibilities, and the precondition ensures that both are legal. If k is chosen, p_k points are earned and the next legal challenge to consider is $k + d_k + 1$. If k is not chosen, no points are earned and the next legal challenge is consider is $k + 1$ — if k was legal, anything after it will be too.
>
> **Final answer.** The first challenge is 0.

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.

Memoization. Identify how to parameterize subproblem state for efficient lookup, typically in an array.

> k is an integer $0..n-1$, so `maxscore(k)` can be stored in `MS[k]`. (Note that a single slot can be used for all of the base cases — store `maxscore(k)` for $k \geq n$ in `MS[n]`.)

Order of computation. Loop(s) can be used to fill in the array; identify whether the loop(s) go from small large index values or vice versa.

> The base case is $k = n$ so `k` needs to be filled in from large to small.

Dynamic programming. Combine the memoization and order of computation with the base and main cases and the top level.

`challenges(d,p)` — Given n challenges, find the challenges to do to maximize the score obtained.

Input: for each challenge i, d_i (number of subsequent challenges missed if challenge i is completed) and p_i (points earned by completing challenge i)

Output: which challenges to complete

Legal solution: if challenge i is completed, no challenge $i + 1 < j < i + d_i + 1$ is completed

Optimization goal: max score

```
// initialize the base case - k ≥ n
MS[n] = 0

// fill in the rest of the array
for k = n-1 downto 0 do
  if k + d_k + 1 ≥ n
    MS[k] = max { MS[n]+p_k , MS[k+1] }
  otherwise
    MS[k] = max { MS[k + d_k + 1]+p_k , MS[k+1] }

// solution
return the challenges for MS[0]
```

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

The running time for a dynamic programming algorithm is the number of slots in the array times the work done per slot to compute the subproblem solution. The time for reconstructing the solution afterwards is the solution length times the work done per slot, so filling in the array dominates.

$k = 0..n$ so `MS` has O(n) slots. Computing `MS[k]` is $O(1)$ — it involves the maximum of two values and a bit of arithmetic. Thus the running time is $O(n)$.

Space is $O(n)$ for the array.