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.

Longest Increasing Subsequence

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.

> Given a sequence S of numbers, find the longest subsequence containing increasing numbers. The numbers in the subsequence must occur in that order in S, but need not be consecutive in S.
>
> Input: sequence S of numbers
>
> Output: (sub)sequence of numbers
>
> Legal solution: elements of the (sub)sequence are in increasing order
>
> Optimization goal: longest increasing subsequence

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

> `5 10 2 7 10 1 18 3`
>
> Both `5 10 18` and `2 7 10 18` are increasing subsequences; `2 7 10 18` is longer.

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 length of S.
>
> The smallest problem is $n = 0$.

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

> This is a variation of a "find subset" problem, and there are 2^n possible subsets — two choices (include or not include) for each element — and $O(2^n)$ possible subsets that are increasing subsequences.

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.

> Since subsequent choices of elements in the subsequence depend only on the last element included so far, this seems like a candidate for dynamic programming being able to improve the running time.

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 subset problem — a subsequence is a subset of the original sequence. There is also an ordering aspect, but the order of the elements included in the subsequence is dictated by their order in the original sequence rather than something that is part of the task to determine.

Process input — for each number, determine whether or not to include it in the subsequence

Produce output — find the next number in the subsequence

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 whole solution is a (complete) subsequence, so a partial solution is (the beginning of) a subsequence.

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 include the current element of S in the subsequence.

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.

Original problem: Given a sequence S of numbers, find the longest subsequence containing increasing numbers. The numbers in the subsequence must occur in that order in S, but need not be consecutive in S.

Subproblem / generalized problem: Given a sequence S of numbers and the beginning of a subsequence of S, find the longest subsequence containing increasing numbers continuing from that point. The numbers in the subsequence must occur in that order in S, but need not be consecutive in S.

Input: sequence S of numbers, subsequence-so-far T, current point k in S

Output: (sub)sequence of numbers

Legal solution: elements of T + the (sub)sequence are in increasing order

Optimization goal: longest increasing subsequence

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

There are no more choices to make when we've reached the end of S — $k = n$. Return an empty sequence and length 0.

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.

The two alternatives are to add the current element of S to the end of the subsequence or not, but adding is only an option if it is bigger than the last element in T.

```
if S[k] is greater than the last element of T then
  // include current element in subsequence
  (S1,len1) ← subseq(S,T with S[k] appended,k+1)
```

```
    // don't include current element in subsequence
    (S2,len2) ← subseq(S,T,k+1)
    if len1 > len2 then
      return (S[k] with S1 appended,len1+1)
    otherwise
      return (S2,len2)
  otherwise
    // don't include current element in subsequence
    return subseq(S,T,k+1)
```

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

> **Initial subproblem.** T is empty and $k = 0$.
>
> **Setup.** Nothing needed.
>
> **Wrapup.** Return just the subsequence found, not the length.

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

> $n = 0$ hasn't been considered — the solution is an empty subsequence.

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.** The length of the sequence remaining to consider is $n - k$, and k increases by 1 with each recursive call.
>
> **Reaching the end.** The base case is when $k = n$, and k increases by 1 each time.

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

> Only legal alternatives were considered, so a complete solution is a legal solution.

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.

> Both alternatives (include or not) are considered, and "include" is only considered if the element is legal to add to the subsequence.
>
> The friends return solutions for the rest of S, so that is appended to the partial solution so far and the alternative we picked.

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 initial subproblem is the original problem, and the result of that includes the subsequence. There is no setup to consider.

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.

The input for the subproblems is the original sequence S, the subsequence-so-far T, and the current position k in S.

> S is the original sequence; it is not part of characterizing particular subproblems and so doesn't need to be accounted for in the memoization.
>
> k is an integer $0..n$.
>
> For T, observe that all that matters for subsequent choices is the last element of T. Also observe that the last element of T (as well as every element of T) is also an element of S, so T could be fully represented for the purposes of subproblem state as the index in S where the element that is last in T is stored. This is also an integer $0..n$.
>
> Let `L[k][t]` be the length of the longest increasing subsequence of `S[k..n-1]` with elements greater than `S[t]`. The initial subproblem is a little problematic because t is the index of an element of T, and T is empty to start. We can write it as `L[0][-1]` and recognize that it isn't stored in the array but is computed the same way as any other subproblem.

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. `L[k][]` only depends on `L[k+1][]` so the order for `t` doesn't matter.

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

> `subseq(S)` — Given a sequence S of numbers, find the longest subsequence containing increasing numbers. The numbers in the subsequence must occur in that order in S, but need not be consecutive in S.
>
> Input: sequence S of numbers
>
> Output: (sub)sequence of numbers
>
> Legal solution: elements of the (sub)sequence are in increasing order
>
> Optimization goal: longest increasing subsequence

```
// initialize the base cases - k=n
for t = 0 to n-1 do
  L[n][t] = 0

// fill in the rest of the array
for k = n-1 downto 0 do
  for t = 0 to n-1 do
    if S[k] > S[t] then    // can include S[k], choose best alternative
      L[k][t] = max(L[k+1][k]+1,L[k+1][t])
    otherwise              // can't include S[k]
      L[k][t] = L[k+1][t]

// solution
return the subsequence for max(L[0][t])  for 0 ≤ t < n
```

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

The array has $(n+1) \times (n+1)$ slots and computing each value is $O(1)$ (two alternatives to consider), so the total running time is $O(n^2)$.