> 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.

Longest Increasing Subsequence

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.

  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.

Identify avenues of attack.

> Determine the algorithmic approach(es) to try.

- *Targets.*

  > Backtracking algorithms are fundamentally exponential; with dynamic programming we hope to reduce the problem to polynomial or pseudopolynomial, but that is not always possible.

- *Paradigms and patterns.*

  Paradigm: dynamic programming.

  > (This is dictated.)

  > Consider the backtracking patterns.

  This is a subset task — which of the elements to include in the sequence.

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

  Produce output: repeatedly find the next element in the subsequence

  Process input: for the next element in S, include it or not?

- *The series of choices.*

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

> The overall running time of a dynamic programming algorithm depends on the size of the array and the time to compute each subproblem's solution, the latter of which depends on the branching factor.
>
> For subset problems, process input results in a branching factor of 2 while produce output has a branching factor of $O(n)$. Let's start with the lower branching factor...

For the next element in S, include it in the subsequence or not?

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 elements of S left to consider.

- *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 elements of S chosen so far.

  - *Alternatives.*

    > What are the legal alternatives for the next choice?

    If the next element is greater than the last one picked, take it or not. Otherwise, the only choice is not to take it.

  - *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.
    >
    > For dynamic programming, the subproblem's output is just the solution for the subproblem (not a complete solution). We also reframe the output to be the value of the solution rather than the specific choices made.

    The subproblem's task is to solve the rest of the problem in light of the choices made so far — find the longest increasing subsequence in the rest of S, given the subsequence T picked so far. Reframing this in terms of the value of the solution means that the task is to find the *length* of the longest increasing subsequence rather than the sequence itself.

    Task: `subseq(`S'`,`T`)` — find the length of the longest increasing subsequence of S' that can follow T

    Input: sequence S' of numbers, increasing subsequence T of numbers

    Output: length of the longest increasing subsequence of S' that can follow T

  - *Memoization.*

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

> The subproblem is defined by S' and T — the remaining part of the original sequence not yet considered and the partial solution (an increasing subsequence). Two parameters means a 2D array. We need to figure out how to represent S' and T as array indexes (integers ≥ 0).

Subproblems are defined by the partial solution (increasing subsequence) and the remaining part of the original sequence not yet considered.

Since we are moving through the original sequence S in order, S' can be represented with an array of all of the items in S and an index k — S' would be the elements $S[k..n-1]$, inclusive.

For T, observe that actually only the last element in T matters for picking the next element. That's a single value, but all we know are that the elements of S are numbers — we don't know that they are integers or ≥ 0. However, also observe that the elements of T are elements of S — so we could represent the last element of T by its index in S rather than its actual value.

We will use `L[k][t]` to store the length of the longest increasing subsequence of $S[k..n-1]$ with elements greater than $S[t]$.

- *Base case(s).*

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

  A complete solution is when all of the elements of S have been considered — $|S'| = 0$ and thus $k = n$. The length of the longest increasing subsequence in this case is 0 because there aren't any elements from which to create a subsequence. `L[n][k] = 0` for $0 \leq k \leq n$.

- *Main case.*

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

  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] > 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]
  ```

- *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.

    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]`.

  - *Setup.*

    > Whatever must happen before the initial subproblem is solved.

    Nothing to do.

  - *Wrapup.*

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

    Nothing to do — the longest increasing subsequence is what we want.

- *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).

- *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.

  > The base case is $k = n$, so the loop for k goes from large $(n - 1)$ to small (0). The order of computation for t doesn't matter because `L[k][...]` is defined in terms of `L[k+1][...]`.

  `algorithm subset(`S`)` — find the longest increasing subsequence of S

  Input: sequence S of numbers

  Output: (sub)sequence of numbers

```
// 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
  for t = 0..n-1
    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]

// compute the subproblems with t=-1  - let L'[k] correspond to L[k][-1]
L'[n] = 0
for k = n-1 downto 0
  L'[k] = max(L[k+1][k]+1,L'[k+1])  // always have option to take when T is empty

return the subsequence for L'[0]
```

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

- *Termination.*

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

  The fill-in-the-array loops are counting loops and the array is finite, so all of the slots will eventually be filled and the loops will end. Reconstructing the solution involves one repetition per step in the solution path, which is also finite.

- *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.

    There aren't any more elements left in the sequence to consider, so there's nothing else to pick. The max length obtainable from no elements is 0.

  - *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.

All legal alternatives are considered: Since the solution is a subsequence of S, every element of S is either in that subsequence or not. The option of including the current element is only considered if it is larger than the last thing in the sequence so far.

The right subproblems: for `L[k][t]`, the two subproblems are `L[k+1][k]` (element $S[k]$ is part of the subsequence) and `L[k+1][t]` (element $S[k]$ is not part of the subsequence). Either way, element $S[k]$ has been considered so we move on the next element, $S[k+1]$. If element $S[k]$ is added to the subsequence it is the next last thing in the subsequence; if it is not, the current last element in the subsequence ($S[t]$) is unchanged.

The friends return the longest sequence resulting from each of the legal alternatives; for the "include $S[k]$" option, add 1 to the length returned to account for $S[k]$. The solution for our subproblem is the better (maximum) of the include and don't include options.

– *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 solution returned is for `L'[0]`, which is corresponds to `L[0][-1]` (the initial subproblem). The subsequence is the desired answer. There is no setup or wrapup to consider.

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.

The array has n^2 slots and computing each value is $O(1)$, so the total running time is $O(n^2)$. The space requirements are also $O(n^2)$.

- *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.

A produce output approach (find the next element to include in the sequence) would have a branching factor of $n - k = O(n)$. The subproblems are the same — still parameterized by the partial solution and the rest of S — so the array is still $n \times n$, resulting in a running time of $O(n^3)$. So, choosing the smaller branching factor was a good idea.