24 Sept 2021

pg 111 of text

```
FastLIS2(A[1..n]):
1:  A[0] = −∞                          ⟨⟨Add a sentinel⟩⟩
2:  for i ← n downto 0
3:      LISfirst[i] ← 1
4:      for j ← i + 1 to n             ⟨⟨...or whatever⟩⟩
5:          if A[j] > A[i] and 1 + LISfirst[j] > LISfirst[i]
6:              LISfirst[i] ← 1 + LISfirst[j]
7:  return LISfirst[0] − 1             ⟨⟨Don't count the sentinel⟩⟩
```

2 to 3 : {2, 3}

4 to 3 : ∅

LISfirst is a 1-d array
↳ stores the length of the LIS starting at that index

How much space? A is $\Theta(n)$

LISfirst[i] allows $i = 0$ through $i = n+1$

↳ is $\Theta(n)$

i, j $\Theta(1)$

∴ In total, $\Theta(n)$

Time Complexity:

1: $\Theta(1)$
2: for $i = n \dots 0$
3: $\quad \Theta(1)$
4: $\quad$ for $i = i+1$ to n
5: $\quad\quad$ 4 lookups + 2 comparisons + 1 add = 6 ops = $\Theta(1)$
6: $\quad\quad$ 5 steps = $\Theta(1)$
7: $\Theta(1)$

($\Theta(n)$ for lines 3–6; $\Theta(n-i)$ for lines 4–6)

Rough sketch: $\Theta(1) + \Theta(n^2) = \Theta(n^2)$

$$\sum_{i=0}^{n} \Theta(1) + \sum_{j=i+1}^{n} \Theta(1) = \sum_{i=0}^{n} (n-i) = \sum_{i=0}^{n} i = \Theta(n^2)$$

$\therefore$ $\Theta(n^2)$ is the time complexity

much better than $\Theta(2^n)$!!

```
FastSplittable(A[1..n]):
  SplitTable[n+1] ← True
  for i ← n down to 1
      SplitTable[i] ← False
      for j ← i to n
          if IsWord(i, j) and SplitTable[j+1]
              SplitTable[i] ← True
  return SplitTable[1]
```

Figure 3.3. Interpunctio verborum velox

(and instructors, and textbooks) make the mistake of focusing on the table—because tables are easy and familiar—instead of the *much* more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recurrence is incorrect, we are well and truly hosed.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

Dynamic programming algorithms are best developed in two distinct stages.

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part. A complete recursive formulation has two parts:

   (a) **Specification.** Describe the problem that you want to solve recursively, in coherent and precise English—not *how* to solve that problem, but *what* problem you're trying to solve. Without this specification, it is impossible, even in principle, to determine whether your solution is correct.

   (b) **Solution.** Give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways your recursive algorithm can call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

→ what are all of the ways that recursive problem can be called
→ in the ~~next~~ 2 steps, each of these calls is like a vertex / node. + we'll build a graph.

(b) **Choose a memoization data structure.** Find a data structure that can store the solution to *every* subproblem you identified in step (a). This is usually *but not always* a multidimensional array.

(c) **Identify dependencies.** Except for the base cases, every subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

(d) **Find a good evaluation order.** Order the subproblems so that each one comes *after* the subproblems it depends on. You should consider the base cases first, then the subproblems that depends only on base cases, and so on, eventually building up to the original top-level problem. The dependencies you identified in the previous step define a partial order over the subproblems; you need to find a linear extension of that partial order. ***Be careful!***

(e) **Analyze space and running time.** The number of distinct subproblems determines the space complexity of your memoized algorithm. To compute the total running time, add up the running times of all possible subproblems, *assuming deeper recursive calls are already memoized*. You can actually do this immediately after step (a).

(f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence, and replacing the recursive calls with array look-ups.

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

3.5 Warning: Greed is Stupid

If we're incredibly lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. Like a backtracking algorithm, a greedy algorithm constructs a solution through a series of decisions, but it makes those decisions directly, *without* solving at any recursive subproblems. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are quite rare. Nevertheless, for many problems that should be solved by backtracking or dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the text segmentation problem might find the shortest (or, if you prefer, longest) prefix of the input string that is

Doesn't have to be optimal!

DAG built where
→ vertices are the nodes
→ edges (directed) point from a node to a node that depends on it

Thm! Given a DAG, there always exists a total order of the nodes compatible w/ it.
"topological sort"

e.g.)

order:
A, B, C
or-
B, A, C
(not unique!)

2(d) Find any one of these compatable ordrs!

Create a DP for los(A)
longest oscillating subsequence

A = input array

oscillating sequence: $x_1 \not< x_2 \not> x_3 \not< x_4 \not> \ldots$ (made cons. w/ book now.)

Ch 6, Q4, part (d)

1st call: los(A)

for the recursive call: you can add parameters!
e.g., booleans, integers, real values,
whatever you need to define/solve the
recursive problem.