Jeff Erickson : Algorithms

Chapter 3. Dynamic Programming. (Part 1)

Example 1. Mātrāvṛtta : the study of poetic meter (prosody 韵律).

- light ~~beats~~ syllable lasts 1 beat
- heavy syllable last 2 beats.

4-beat meters: — —, — • •, • — •, • • —, • • • •

↑ Long ↑ Short.

Observation: number of meter of n beats is the sum of ∼ n-1 and ∼ n-2.

$$M(n) = M(n-1) + M(n-2),$$

with base case $M(0) = 1$. $M(1) = 1$.

↳ shows that $\Theta(2^n)$.

Memoization: Remember everything (if undef, fill in).

Dynamic programming. filling tables deleberately.

↖ planning or schedules

Example 2. Interpunctio Verborum Redux.

Given a string $A[1..n]$ and a subroutine IsWORD that determines whether a given string is a word.

Ask whether A can be partitioned into a seq of words.

▷ Define a function, Splittable (i) returns true ⟺ the suffix $A[i..n]$ can be partitioned into seq of words.

▷ Need: Splittable (1).

$$\text{Splittable}(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \left(\text{IsWord}(i,j) \wedge \text{Splittable}(j+1) \right) & \text{o.w.} \end{cases}$$

Directly impl will cause $O(2^n)$ in the worst case.

- Memorize function Splittable into array Splittable[1..n+1].
- Each subprob Splittable(i) depends only on res of subprobs Splittable(j) ($j>i$).
- fills the array in decr indx order.

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

1. The pattern: Smart Recursion.
   - Recursion without repetition

> Dynamic Programming is <u>not</u> filling in tabls.
> It's about Smart Recursion!

▷ Processes

I. Formulate the problem <u>recursively</u>.

repr your answer ~~by~~ smaller subprobs.

- Specification — what's the problem going to solve
- Solution — clear recursive formula or algo for the whole prob in terms of <u>exactly</u> the same prob.

II. Build sols to your recur. from bottom up.

- Identify subprobs
- Choose a memoization data structure
- Identify dependencies
- Find a good eval order <u>Be careful!</u>
- Analyze space & Running Time

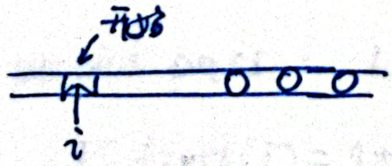
□

Example 3. Longest Increasing Subseq.

- The recur. form.

LISbigger (i,j) := length of LIS of $A[j \dots n]$ every elem is larger than $A[i]$.

$$\text{LISbigger}(i,j) = \begin{cases} 0 & j>n \\ \text{LISbigger}(i,j+1) & A[i] > A[j] \\ \max\begin{cases} \text{LISbigger}(i,j+1) \\ \text{LISbigger}(j,j+1)+1 \end{cases} & \text{o.w.} \end{cases}$$

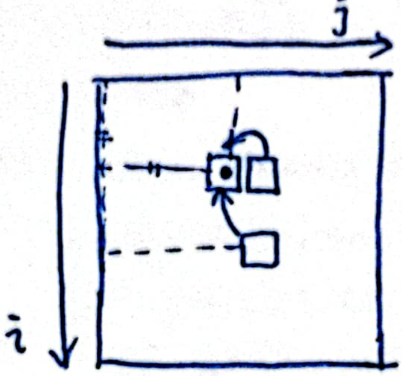


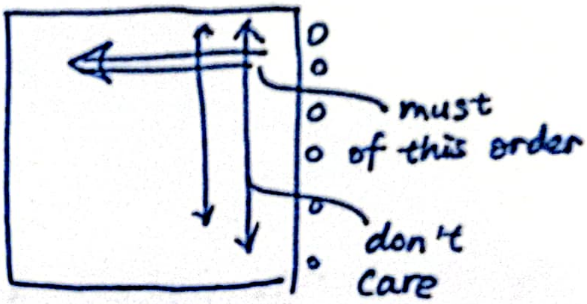
▷ Only $O(n^2)$ distinct subproblems to consider.

▷ Store it in 2D array LISbigger $[0..n, 1..n]$.

- Evaluation order

  ▷ Each entry LISbigger[i,j] is filled <u>after</u> ~[i, j+1]
  ~[j, j+1]






```
FASTLIS(A[1..n])
    A[0] ← -∞
    for i ← 0 to n
        LISbigger[i, n+1] ← 0        } base case.
    for j ← n downto 1
        for i ← 0 to j-1
            Keep ← 1 + LISBIGGER[j, j+1]
            Skip ← LISBIGGER[i, j+1]
            if A[i] ≥ A[j]
                LISBIGGER[i,j] ← Skip
            else
                LISBIGGER[i,j] ← max{Keep, Skip}.
    return LISbigger[0, 1]
```

- The second pass : LISFirst(i) := LIS begin with A[i]

  i.e. $\text{LISfirst}(i) = 1 + \max\left\{\text{LISFirst}(j) : \begin{matrix}(j>i)\wedge \\ (A[j]>A[i])\end{matrix}\right\}$

  as $\max \emptyset = 0$.

  and $A[0] \leftarrow -\infty$.

· dependency: LISFirst[i] depends only on LISFirst[j] as $j>i$.

```
FASTLIS2(A[i..n]):
    A[0] = -∞
    for i ← n downto 0
        LISFirst[i] ← 1
        for j ← i+1 to n
            if A[j] > A[i] and 1+LISfirst[j] > LISFirst[i]
                LISFirst[i] ← 1+LISfirst[j]
    return LISfirst[0]-1 .
```

~~Example 4. Edit distance.~~