Algorithms

Dynamic Programming (part 2)

Notes

- Oral grading - make sure you have a spot!
  Also: please avoid class-mates after you present.

- HW0 is graded. + in BB
  - #4: one part was extra credit (except for grad students)

  max: 50 (for ugrads)
  60 (for grads)

  Min: 22
  Max: 52
  Average: 37.33
  (Fairly low - but don't worry yet!)

Recap: Backtracking

- Find a small choice that reduces the problem size
- For each answer to the choice, choose answer + recurse

  (while considering only subsolutions consistent with that choice)

Next: Dynamic Programming

Dynamic programming is just smart recursion.

- Recurse - don't repeat

Often computed values are stored in some table for later lookups

- or -

Can rearrange to fill table from ground up.

Does assume fcn(i) always returns same value.

Note: This takes up more space.

Last time: Fibonacci #5

```
FIB(n):
if n<2:
    return n
else
    return FIB(n-1) + FIB(n-2)
```

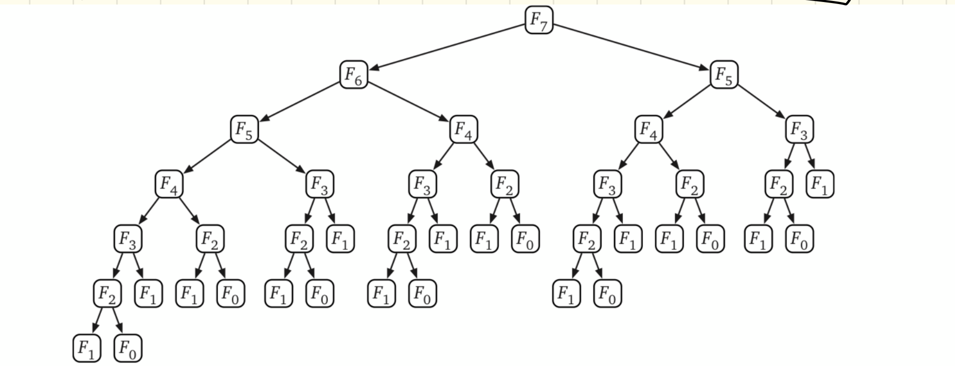


Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

But this is dumb!

F_4 is always the same – why recompute?

Better:

```
MEMFIBO(n):
   if (n < 2)
        return n
   else
        if F[n] is undefined
             F[n] ← MEMFIBO(n − 1) + MEMFIBO(n − 2)
        return F[n]
```

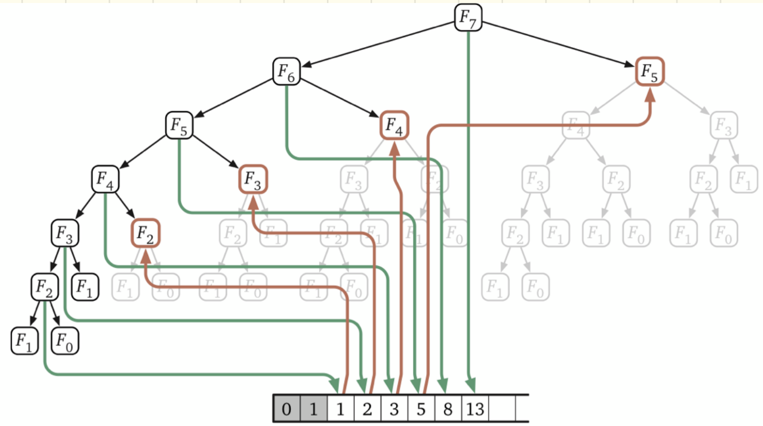


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Steps:

① Formulate the recursion

② Build solution from base case up.

- identify subproblems
- identify dependencies:
  ie: F(6) depends on F(5) + F(4)
- choose data structure
  ie: often array, 1d or 2d, or even a few variables
- choose evaluation order

- write pseudo code, then analyze time/space

Let's look at an old friend or two...

Text Segmentation:

Idea: Given a string of "letters", break into "words".

Assume: Given ISWORD(w), which takes a string & says true or false

O(1) time

Backtracking:

Starting at beginning, check every prefix:

if ISWORD(A[1]), recurse on A[2...n]

if ISWORD(A[1,2]), try A[3..n]

if ISWORD(A[1,2,3]), try A[4..n]

⋮

if ISWORD(A[1..i]), try A[i+1, n]

if ISWORD(A[1..n]), done

If any succeed, return true

Recursion: set up a function

$$\text{Splittable}(i) := \begin{cases} \text{true} & \text{if } i > n \\ \underset{\text{OR}}{\bigvee_{j=i}^{n}} \left(\text{IsWord}(i,j) \overset{\text{and}}{\wedge} \text{Splittable}(j+1) \right) \end{cases}$$

Code:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Idea: to improve, think about calls:

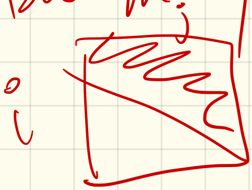
```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if IsWord(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Each splittable(i) is called many times.
(Same for IsWord(i, j))
Just memoize them!

How many? (too many

I need SPLITTABE(i) for every i from 1 to n
→ remember them

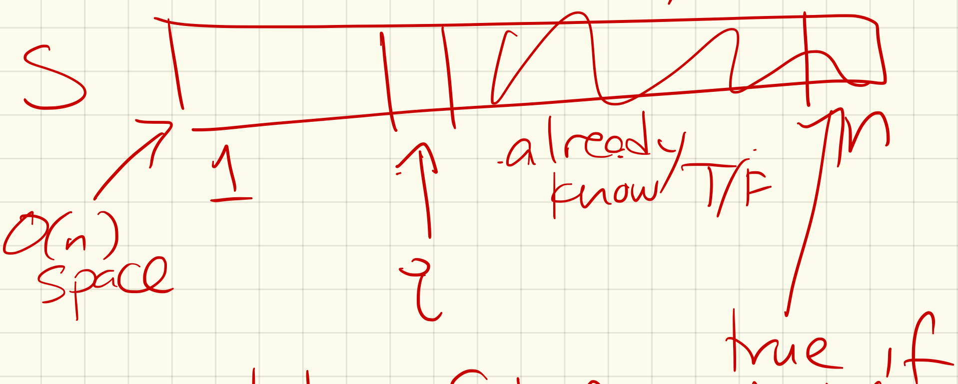
How many IsWord(i,j)'s?



$O(n^2)$

Translate to a loop:

I can fill in position n immediately



initialize S to false

for $i \leftarrow n-1$ to 1

$O(n-i)$ for $j = i$ to n

$O(1)$ if isWord(i, j) and $S[j] ==$ true

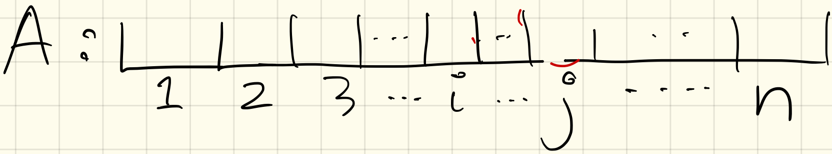
$S[i] \leftarrow$ true

$$\sum_{i=1}^{n-1} (n-i) = O(n^2)$$

Back to LIS:

Some notation:

Let $LIS(i,j) :=$ length of longest subsequence of $A[j..n]$ with elements $> A[i]$

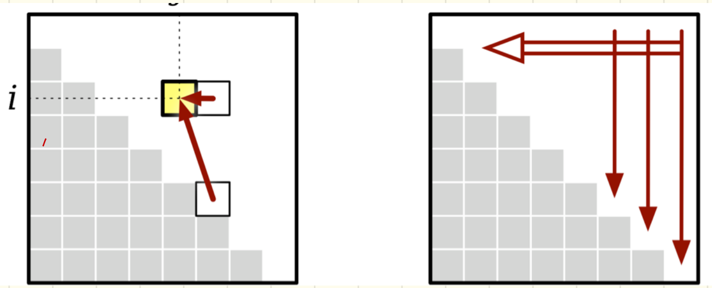


Then:

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$

What are my dependencies?

So, build a solution:



$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i, j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

Algorithm:

```
LIS(A[1..n]):
    A[0] ← −∞                                ⟨⟨Add a sentinel⟩⟩
    for i ← 0 to n                           ⟨⟨Base cases⟩⟩
        LIS[i, n+1] ← 0

    for j ← n downto 1
        for i ← 0 to j − 1
            if A[i] ≥ A[j]
                LIS[i, j] ← LIS[i, j+1]
            else
                LIS[i, j] ← max{LIS[i, j+1], 1 + LIS[j, j+1]}

    return LIS[0, 1]
```

Time & Space: