Algorithms

Backtracking
Dynamic Programming

Recap:

- Sign up for HW2 grading
  (We'll end 5 min. early...)
- Perusall due Wed. (& likely Friday)
- Gradebook note:
  use blackboard

Longest Increasing Subsequence or (LIS)

Given: List of integers $A[1..n]$

Goal: Find longest subsequence whose elements are strictly increasing

Formally: $A[1..n]$, Find largest k s.t. $1 < i_1 < \cdots < i_k \le n$

s.t. $A[i_j] < A[i_{j+1}]$ for every j

Example:

$$[12, 5, \underset{3}{\underline{1}}, \underset{4}{\underline{3}}, \underset{5}{\underline{4}}, 13, \underset{7}{\underline{6}}, \underset{8}{\underline{11}}, 2, \underset{10}{\underline{20}}]$$

IS: 12, 13, 20

Best? length 6 $k=6$ in ex

Formalize (a la backtracking):

The LIS of A[1..n] is either:

- the LIS of A[2..n] (skip #1)
- A[1] followed by LIS of A[2..n] (include #1)
  where everything is > A[1]

(or is it?)

Go back to that example...

added a param to my fcn

Let:

LISBIGGER(i, j) :=
Longest subsequence from $A[j..n]$
with all elements $> A[i]$

Then: backtracking recursion

$$\text{LISBIGGER}(i,j) = \max \begin{cases} \text{Base case: if } j > n\text{: return } \{\} \\ \text{Include } A[j]\text{: if } A[i] < A[j]\text{, could be } 1 + \text{LISBIG}(j, j+1) \\ \text{Not include } A[j]\text{: } \text{LISBIG}(i, j+1) \end{cases}$$
$(i < j)$

LIS $(A[1..n])$:
adds $-\infty$ at $A[0]$
return (LISBIGGER$(0, 1)$) -1

Nicer picture:

must skip A[j]

if A[j] > A[i], could include or not

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

Alternatively, if you prefer pseudocode:

```
LISBIGGER(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISBIGGER(i, j + 1)
  else
      skip ← LISBIGGER(i, j + 1)
      take ← LISBIGGER(j, j + 1) + 1
      return max{skip, take}
```

Runtime: Let $L(n)$ = runtime on array of size n

Then: $L(n) \leq 2L(n-1) + 1$

$= O(2^n)$

Sec. 2.7 : (take 2)

Another approach:

Last version considered the input A one letter at a time.

Alternative: build output one at a time.

Given a position i, construct LIS in $A[i..n]$. (which includes $A[i]$.

Then: LISFIRST(i) :=

$$1 + \max_{j>i} \{ \text{if } A[j] > A[i]: \text{LISFIRST}(j) + 1 \}$$

Pseudocode: To solve LIS, use our helper function.

```
LISFIRST(i):
    best ← 0
    for j ← i + 1 to n
        if A[j] > A[i]
            best ← max{best, LISFIRST(j)}
    return 1 + best
```

try all $j > i$ & recurse on any viable next values ($A[j] > A[i]$)

```
LIS(A[1..n]):
    best ← 0
    for i ← 1 to n
        best ← max{best, LISFIRST(i)}
    return best
```

```
LIS(A[1..n]):
    A[0] ← −∞
    return LISFIRST(0) − 1
```

how to call helper fun

Pausing for a moment:

- Done with recursion (for now)
  (skipping binary tree section)
- Chapter 3: dynamic programming
- Chapter 4: Greedy algorithms

Then graphs after that
(for a while)

Midterm: likely week of
Oct. 14 or / Oct 7.

(More to come soon...)

Dynamic Programing

- a fancy term for smarter recursion:

memoization

- Developed by Richard Bellman in mid-1950s

("programming" here actually means planning or scheduling)

Key: When recursing, if many recursive calls to overlapping subcases, remember prior results and don't do extra work!

Simple example:

Fibonacci Numbers

$F_0 = 0, \; F_1 = 1, \; F_n = F_{n-1} + F_{n-2}$
$\forall n \geq 2$

0, 1, 1, 2, 3, 5, 8, 13, ...

Directly get an algorithm:

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

Runtime:

$$F(n) = 1 + F(n-1) + F(n-2)$$

exponential:

$O(\phi^n)$ exponential

Applying memoization:

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

First time F[19] is called, actually recurse + store answer.

All later calls: look it up.

Better yet:

<u>ITERFIBO</u>(n):

$\Theta(n)$

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

for $i \leftarrow 2$ to n

$\Theta(1)$ $F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

Correctness:

Run time & space

$\Theta(n)$

need array of size n

Even better!

```
ITERFIBO2(n):
    prev ← 1
    curr ← 0
    for i ← 1 to n
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

Run time/space:

$O(n)$ time

$O(1)$ space

Note: We'll skip 3.2, although you're welcome to read! :)

Next:

- Text segmentation
- Longest increasing subsequence

Key: