Tuesday March 24, 2015

Dynamic Programming:

Simple example: computing Fibonacci numbers:

$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

defined by:

$F_0 = 0,\ F_1 = 1,$

and for $n > 1$: $F_n = F_{n-1} + F_{n-2}$

Natural algorithm:

Fib1(n):

```
    if n=0, return(0)
    if n=1, return(1)
    return( Fib1(n-1)+Fib1(n-2))
```

What's running time?

Look at $T(n)$ = # of steps for computing n^{th} Fibonacci #

$T(0) = O(1)$

$T(1) = O(1)$

for $n > 1$: $T(n) = T(n-1) + T(n-2) + O(1)$

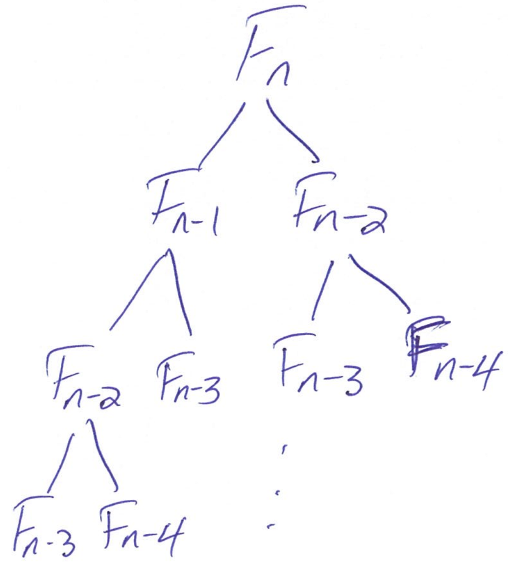
Then $T(n) \geq F_n$

and F_n is HUGE

$F_n \approx \frac{\phi^n}{\sqrt{5}}$ where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the "golden ratio"

So exponential-time algorithm.

Why is Fib1 so slow?



Recomputing many times the answer to small subproblems.

Better approach: Only compute the answer to each subproblem once. To ensure this, start with smallest $\longrightarrow$ largest (bottom-up approach)

Fib2(n):

```
    if n=0, return(0)
    if n=1, return(1)
    create an array F[0...n]
    F[0]=0, F[1]=1
    for i=2→n
        F[i]=F[i-1]+F[i-2]
    return(F[n])
```

Running time: $O(1)$ per i so $O(n)$ total time.

Dynamic Programming approach:

1) Define subproblem in words, for example:

$$F(i) = i^{th} \text{ Fibonacci number}$$

2) State a recurrence in terms of smaller subproblems,

example: $F(i) = F(i-1) + F(i-2)$

3) Solve subproblems from smallest to largest.

Longest increasing subsequence:

Input: n numbers $a_1, a_2, \ldots, a_n$

example: 5, 2, 8, 6, 3, 6, 9, 7

a subsequence is a subset in order

example: Using indices 2, 4, 5, 7

gives 2, 6, 3, 9

So a subsequence is a subset $a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}$

where the indices satisfy:

$$1 \leq i_1 < i_2 < \cdots < i_\ell \leq n$$

(So increasing indices)

example: $i_1 = 2 < i_2 = 4 < i_3 = 5 < i_4 = 7$

A subsequence is increasing if

$$a_{i_1} < a_{i_2} < \cdots < a_{i_\ell}$$

example: $a_2, a_5, a_6, a_7 = 2, 3, 6, 9$
is an increasing subsequence since $2<3<6<9$.

Goal: Given $a_1, \ldots, a_n$ find an increasing subsequence of max length.

First, let's focus on finding just the length of the longest increasing subsequence (LIS).

First step: define the subproblem in words.

Natural idea:

$S(j)$ = length of longest increasing subsequence in $a_1, \ldots, a_j$

Goal: compute $S(n)$.

How to write a recurrence for $S(j)$ in terms of $S(1), S(2), \ldots, S(j-1)$?

For example, in our earlier example: 5, 2, 8, 6, 3, 6, 9, 7

$S(1)=1, S(2)=1, S(3)=2, S(4)=2, S(5)=2$

Can we figure out $S(6)$ just from $S(1), \ldots, S(5)$ & $a_1, \ldots, a_5$?

$S(5)=2$ but can we add 6 onto it?

if it corresponds to 2, 3 then yes

but if it corresponds to 5, 8 then no

So we need track of <u>all</u> possible endings

Cleaner approach:

add extra condition to the definition of the subproblem to remember what number it ends at.

Let $L(j)$ = length of longest increasing subsequence in $a_1, \ldots, a_j$ which ends at a_j & includes a_j

Goal: Compute $\max_j L(j)$.

$$L(j) = 1 + \underbrace{\max_i \{L(i) : i < j, a_i < a_j\}}$$

means maximize $L(i)$ where the variable is i & try those i where $i < j$ and $a_i < a_j$

LIS(A):

```
input: A=[a_1,...,a_n]
for j=1 -> n
    L(j)=1, prev(j)=NULL
    for i=1 -> j-1
        if L(i)+1 > L(j) & a_i < a_j
            then L(j)=L(i)+1
                 Prev(j)=i
Let max=1
for i=1 -> n
    if L(i) > L(max) then max=i
Return(L(max))
```

Running time:

$O(1)$ per i

$O(n)$ sized loop over i

$O(n)$ sized loop over j

$O(1) \times O(n) \times O(n) = O(n^2)$ total time.

How to find the actual subsequence?

Keep track of the next to last index i that gives the max

then backtrack

Earlier example:

$A =$	5	2	8	6	3	6	9	7
$L =$	1	1	2	2	2	3	4	4
Prev =	NULL	NULL	1	1	2	5	6	6

to reconstruct $S(7)$ follow the path

$$a_2 \Leftarrow a_5 \Leftarrow a_6 \Leftarrow a_7$$

$$2 \Leftarrow 3 \Leftarrow 6 \Leftarrow 9$$

So it's: 2, 3, 6, 9

To get the subsequence add:

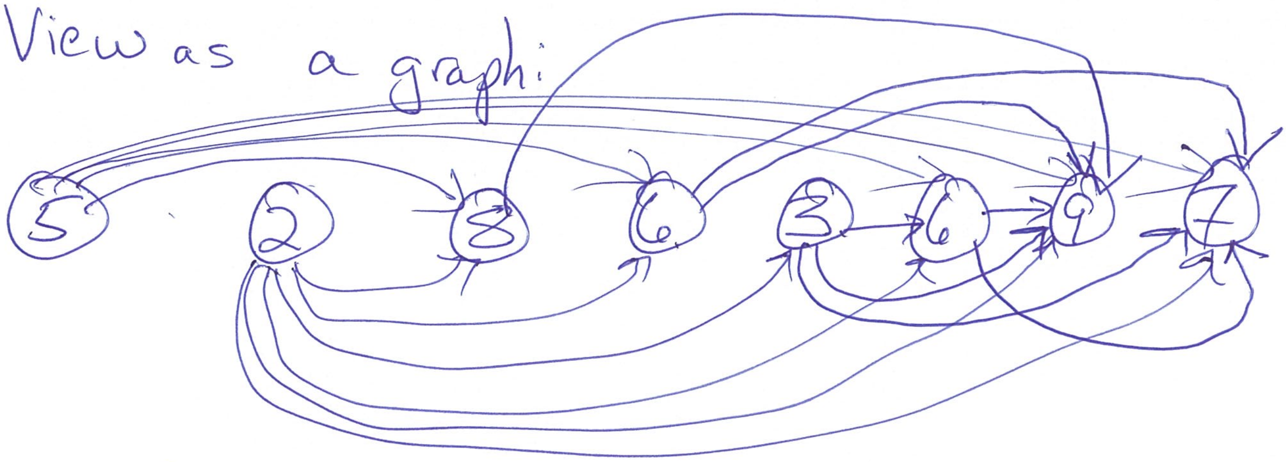
```
i = max
output(i)
while prev(i) ≠ NULL:
    i = prev(i)
    output(i)
```

Key ideas:

- Use prefix of input for subproblems
- strengthen subproblem by adding extra info into it.

Earlier example: 5, 2, 8, 6, 3, 6, 9, 7

View as a graph:



edge from $i \to j$ if $i < j$ & $a_i < a_j$

then $L(j)$ = length of longest path ending at a_j

this graph is a DAG = directed acyclic graph

in topological order (all edges go left to right)

We are finding the longest path in a DAG.