Review: $O()$ notation.

For functions $f(n)$ & $g(n)$,

$f(n) = O(g(n))$ if there is a constant $c > 0$ where $f(n) \leq g(n)$.

Examples:

a) $f(n) = 3n^2 + 10n - 5n^{2.5} + 1.7n^3$

$f(n) = O(n^3)$ and $f(n) = O(n^5)$

b) $g(n) = 5\log^2 n + 7\sqrt{n}$

$g(n) = O(f(n))$

c) $f(n) = n^2$, $g(n) = 2^{4\log n}$

if don't specify base then base 2

So: $2^{\log n} = n$

$\ln n = \log_e n$.

Note, $\log_{10} n = O(\log n)$

$\ln n = O(\log n)$

Manipulating log's:

$g(n) = 2^{4\log n} = \left(2^{\log n}\right)^4 = n^4$

hence, $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$.

How about $f(n) = 3^{\log_5 n}$

express as a polynomial n^c

for constant $c > 0$.

$3^{\log_5 n}$

want to match

so note $3 = 5^{\log_5 3}$

then: $f(n) = 3^{\log_5 n} = \left(5^{\log_5 3}\right)^{\log_5 n} = \left(5^{\log_5 n}\right)^{\log_5 3}$

$= n^{\log_5 3}$

$\log_5 3 < 1$ so $f(n) = O(n)$.

Dynamic Programming (DP):

Toy example: Computing Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

defined by:

$F_0 = 0$, $F_1 = 1$

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

Natural recursive algorithm:

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

Running time???

Look at time as a function of input size n

in this case $n = n^{\text{th}}$ Fib #

other examples: n = # of numbers to sort

n = # of vertices in input graph

n = # of bits in input numbers to multiply

Look at running time in $O()$
ignore constant factors since depend on specific machine.

Model of computation:

- $O(1)$ time to add/subtract/multiply/divide
  any basic arithmetic operation

- unlimited memory
  & $O(1)$ time to read/write unit of memory.

Analyzing Fib1(n):

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

$T(0) = O(1)$ & $T(1) = O(1)$

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

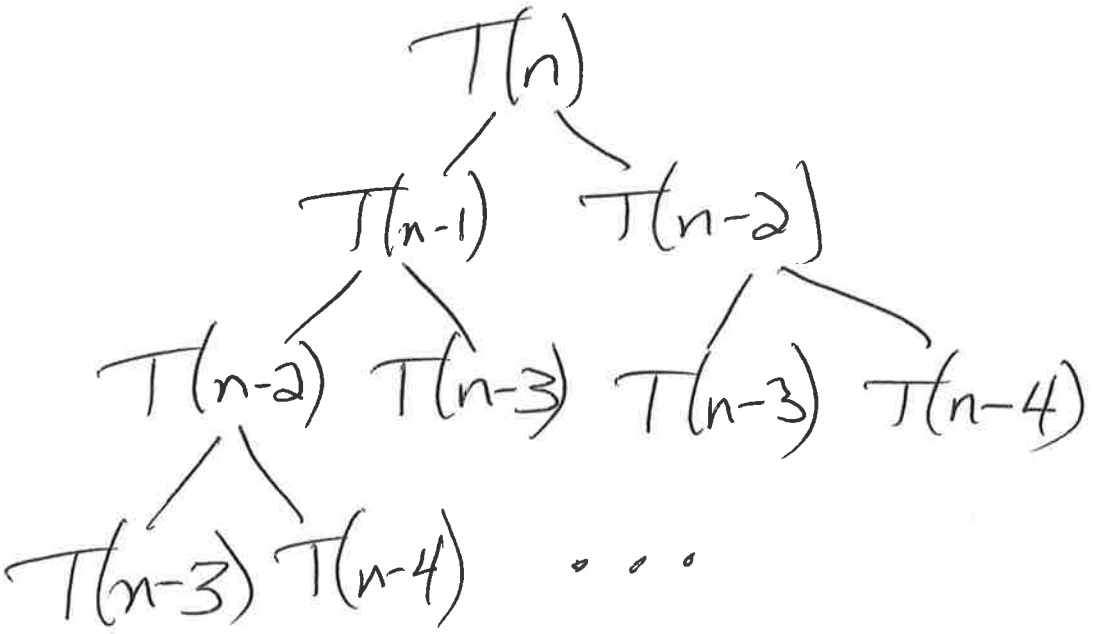
hence, $T(n) \geq F(n)$.

$T(n) = T(n-1) + T(n-2) + c$, $F(n) = F(n-1) + F(n-2)$

$F(n) \approx \frac{\phi^n}{\sqrt{5}}$ where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio.

So exponential-time algorithm.

Why is Fib1( ) so slow?



— Recomputing small subproblems many times.

Better approach:

Work bottom-up.

Start with smallest F(0) & F(1).

& go to larger

then only solve each subproblem once.

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 & $O(n)$ sized loop over i;

$\Rightarrow O(n)$ total time.

Non-trivial example: Longest increasing subsequence (LIS).

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

length of longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}$

which is increasing $a_{i_1} < a_{i_2} < \cdots < a_{i_\ell}$

note, a subsequence is a subset of indices in order:

$1 \le i_1 < i_2 < i_3 < \cdots < i_\ell \le n$.

Example: A = 5, 2, 8, -1, 6, 3, 6, 9, 2, 4, -3, 7

LIS = -1, 3, 4, 7 (or 2, 3, 6, 7, etc.)

length = 4

First step:

Define the subproblem in words

(think inductive proof: define inductive hypothesis)

Natural idea: let $S(j)$ = length of LIS in $a_1, \ldots a_j$

Same problem on prefix of input

(always use prefixes as 1st attempt)

Goal: compute $S(n)$.

Second step:

Write recurrence for $S(j)$ in terms of $S(1), \ldots, S(j-1)$

idea: $S(j)$ is max of $S(j-1)$ and $1 +$ best of $S(j-1)$ ending at $< a_j$

$S(j-1)$: best LIS in $a_1, \ldots, a_{j-1}$

best of $S(j-1)$ ending at $< a_j$: best LIS that can add a_j to.

Example:

i =	1	2	3	4	5	6	7
A =	5	2	8	-1	6	3	6, 9, 2, 4, -3, 7
S =	1	1	2	2	2	2	?

for $S[7]$ how do we know what $S[6]$ ends at?

many possible endings

want smallest ending character

which is 3 from -1, 3

but for $S[4] = 2$ corresponding to 5, 8

so how do we have want LIS ending at -1.

Possible ending numbers are the numbers appearing in A.

For each ending number what's LIS ending at it.

Let $L(j)$ = length of LIS in $a_1, \ldots a_j$ ends at a_j & includes a_j.

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

Recurrence:

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

(for a_j; length of LIS ending at a_i for $a_i < a_j$ & $i<j$)

LIS(A):

```
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 $i \times O(n)$ sized loop over $i \times O(n)$ loop over j

$$O(1) \times O(n) \times O(n) = O(n^2)$$

Earlier example:

A = 5, 2, 8, -1, 6, 3, 6, 9, 2, 4, -3, 7

L = 1, 1, 2, 1, 2, 2, 3, 4, 2, 3, 1, 4

How to find LIS?

keep track of i that achieves the max

Using prev(j)

& then backtrack

So, -1, 2, 4, 7.