Announcements

- HW due Friday

- Next HW out Friday, due the Monday after break.

- For dyn. Programming, most topics will come from lecture notes rather than text.
A Key Fact:

Greedy algorithms (almost) NEVER Work!

(Tattoo this on your hand somewhere...)
Fibonacci Numbers

(A great example of "smart" recursion)

\[ F_0 = 0, \quad F_1 = 1 \]

\[ F_n = F_{n-1} + F_{n-2} \quad {\text{for}} \quad n > 1 \]

\[ F_2 = 1 \quad F_3 = 2 \quad F_4 = 3 \quad F_5 = 5 \quad F_6 = 8 \]

\[ F_n \approx \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{\sqrt{5}} \quad (2) \]

\[ \phi^n \quad \text{golden ratio} \approx 1.6... \]
Pseudo code

\[ \text{RecFibo}(n) : \]
\[ \text{if } n < 2 \]
\[ \quad \text{return } n \]
\[ \text{else} \]
\[ \quad \text{return } \text{RecFibo}(n-1) + \text{RecFibo}(n-2) \]

Runtime? BAD

\[ T(n) = T(n-1) + T(n-2) + O(1) \]
\[ \approx O(\phi^n) \text{ exponential} \]
Why so slow?

# leaves = $2^n$

depth $n$
So take a step back, and use smart recursion.

Why are we recomputing $F(n-2)$ twice?
or $F(n-3)$ 3 times?
or $F(0) + F(1)$ a total of $2^n$ times?!

Try to eliminate redundant calls.
Store result of 1st time any $F(i)$ is computed!
Pseudo code

IterFibo(n):

\[
\begin{align*}
F[0] & \leftarrow 0 \\
F[1] & \leftarrow 1 \\
\text{for } i \leftarrow 2 \text{ to } n \text{ do} & \\
F[i] & \leftarrow F[i-1] + F[i-2] \\
\text{return } F[n]
\end{align*}
\]

time? \(O(n)\) (just 1 for loop)
if addition takes \(\log n\) time, \(O(n \log n)\)

space? \(O(n)\) - a single array of length \(n\)

\text{can improve}
Even Better:

Iter-Fibo 2(n):

if n = 0 or 1
return n
else
  prev ← 0
  curr ← 1
  for i ← 2 to n
    next ← prev + curr
    prev ← curr
    curr ← next
  return curr

Runtime: Still O(n)

Space: O(1)
Dynamic Programming:
Recursion without repetition

1) Formulate the problem recursively

2) Build solutions from the “bottom up”
   - Usually need some data structure to store subproblem solutions
   - Identify dependencies between subproblems

Note: For these, I expect you to analyze running time and space!
Weighted Interval Scheduling (Ch. 6.1)

Set of intervals, want to choose a subset so that no 2 overlap.

Now, however, each interval has a weight, and we want our subset to have maximum possible weight.

Ex:

```
weight = 1

weight = 3

weight = 1
```
Recursive strategy

Jobs $J = \{ J_1, \ldots, J_n \}$
Sort jobs by finish time, and consider last interval.

It is either in the solution, or not.

Suppose $J_n \in \text{OPT}$, then erase conflicts and recurse on $J_1, \ldots, J_{n-1}$.

Suppose $J_n \notin \text{OPT}$, then recurse on rest.
Recursive strategy: $J = \{ j_1, \ldots, j_n \}$

$$\text{OPT}(J) = \max \sum \left[ \omega(j_n) + \text{OPT}(J - \text{jobs that overlap } j_n) \right]$$

Key idea: $j_n$ belongs in OPT

$$\implies \omega(j_n) + \text{OPT}(J - \text{conflicts}) \geq \text{OPT}(\{ j_1, \ldots, j_{n-1} \})$$
Pseudo code

$\text{Compute-Opt}(\mathcal{E}J_1, \ldots, J_n)$

If $n = 0$
    return 0
else
    option 1 ← $\text{Compute-Opt}(\mathcal{E}J_1, \ldots, J_{n-1})$
    $I$ ← set of jobs not overlapping $J_n$
    option 2 ← $w(J_n) + \text{Compute-Opt}(I)$
    return max(option 1, option 2)

$I = \mathcal{E}J_1, \ldots, J_n$
Correctness: Induction on # of jobs

BC: By definition, OPT(1) = 0.

IH: Alg finds optimal soln for \(< j\) jobs.

IS: Consider \(j\) jobs.

We calculate only 2 possibilities - either

\(J_j\) is in OPT or it isn't.

Since our algorithm correctly computes

\(OPT(\overline{U_1 \ldots J_{j-1}}) + OPT(\overline{U_1 \ldots J_{j-1}})\) (since \(|J_j| < |J_{j-1}|\),

we get correct value for \(OPT(\overline{U_1 \ldots J_{j-1}})\).
Runtime \( J(n) \leq 2(J(n-1)) + O(n) \)

worse than Fibonacci! \( \approx n2^n \) (?)

How to improve?

Remove redundant calls.
```
DP-Compute-OPT(J[1..n], j, M[1,...,n]):

If j = 0
    return 0
else if M[j] is not empty
    return M[j]
else
    T ← set of jobs not overlapping J[j]
    option 1 ← DP-Compute-OPT(J[1..j-1])

    option 2 ← w(J[j]) + DP-Compute-OPT(T)

    M[j] ← max(option 1, option 2)

    return M[j]

Initial call will be DP-Compute-OPT(J, n, M)
```
Alternatively, could use a for loop. Would fill in M starting at M[i, j]. At each stage, need to check M[j-1][i] and M[i][i].

\[ \text{space: } O(n) \] (can't use Fib. trick since I is not consistent)
Correctness - same as dumb recursive version.

(also - sort) \( O(n \log n) \)

Runtime: We have an array \( M \) of size \( n \).

At end, we will fill in each entry,

-> For each entry, we do 2 table lookups
t + an addition + a max comparison

\( \Rightarrow O(1) \). (Actually \( O(n) \) to compute \( t \).

Each entry only gets filled in once.

\( \Rightarrow O(n^2) \)