CS 314 - Dynamic Programming

Note Title 2/24/2010

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!

(Tatoo this on your hand somewhere...)

Fibonacci Numbers

(A great example of "smart" recursion)

$F_0 = 0, \quad F_1 = 1$

$F_n = F_{n-1} + F_{n-2} \quad \forall n > 1$

$F_2 = 1$
$F_3 = 2$
$F_4 = 3$
$F_5 = 5$
$F_6 = 8$
$\vdots$

$$F_n = \frac{(1+\sqrt{5})^n + (1-\sqrt{5})^n}{\sqrt{5}} \quad (?)$$

$\approx O(\phi^n)$ ← golden ratio $\approx 1.6\ldots$

Pseudocode

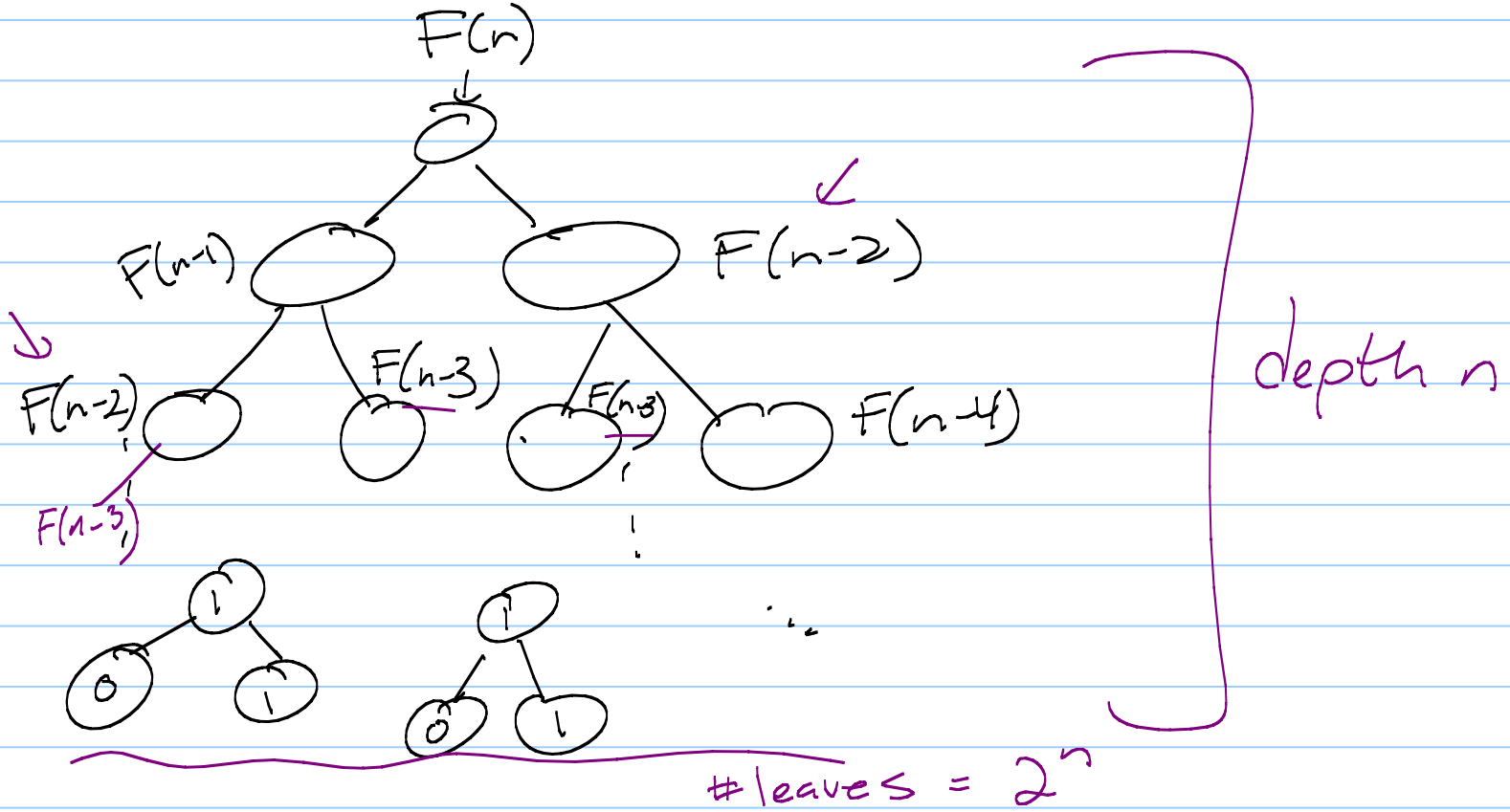
```
RecFibo(n):
  if n < 2
      return n
  else
      return RecFibo(n-1) + RecFibo(n-2)
```

Runtime? BAD

→ $T(n) = T(n-1) + T(n-2) + O(1)$

$\approx O(\phi^n)$ exponential

Why so slow?



So take a step back, & 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):
    F[0] ← 0
    F[1] ← 1
→   for i ← 2 to n
        F[i] ← F[i-1] + F[i-2]
    return F[n]
```

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

↑ can improve

Even Better:

IterFibo2(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 struc. to store subproblem solutions
- identify dependencies b/t subproblems

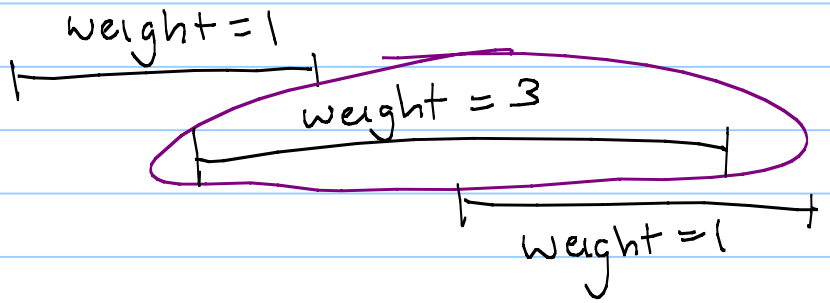
Note: For these, I expect you to analyze running time and space!

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 also, & we want our subset to have maximum possible weight.

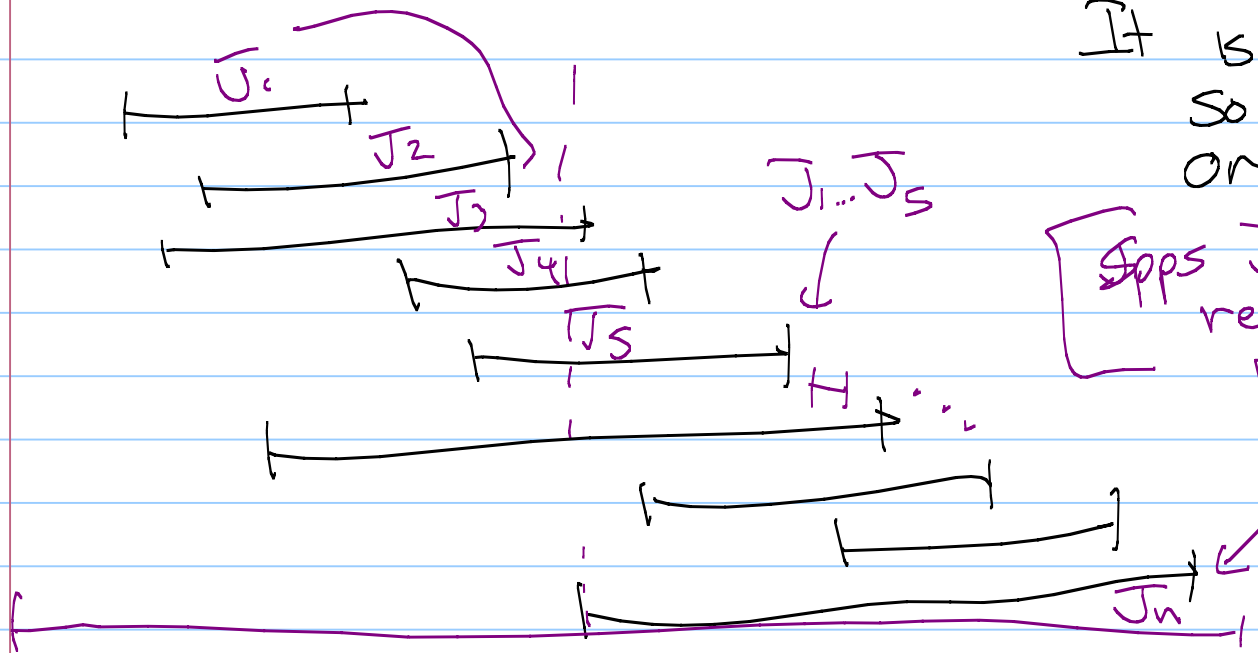
Ex:



Recursive strategy

Jobs $J = \{j_1, \ldots, j_n\}$

Sort jobs by finish time, & consider last interval.



It is either in the solution, or not.

Supp $J_n \notin OPT$. recurse on $J_1 \ldots J_{n-1}$.

Suppose $J_n \in OPT$ then erase conflicts & recurse on rest.

Recursive strategy : $J = \{j_1, \ldots, j_n\}$

$$OPT(J) = \max \begin{cases} OPT(\{j_1, \ldots, j_{n-1}\}) \\ w(j_n) + OPT(J - \{\text{jobs that overlap } j_n\}) \end{cases}$$

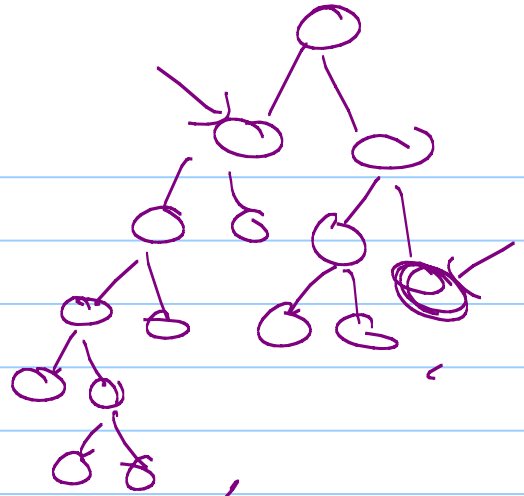
Key idea: j_n belongs in OPT

$$\Leftrightarrow \quad w(j_n) + OPT(J - \text{conflicts}) \geq OPT(\{j_1 \ldots j_{n-1}\})$$

Pseudo code

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

If $n = 0$
    return 0
else
    option1 ← Compute-Opt($\{J_1, \ldots, J_{n-1}\}$)
    I ← set of jobs not overlapping J_n
    option2 ← $w(J_n)$ + Compute-Opt(I)
    return max(option1, option2)



$|I| = n-1$

$I = \{J_1, \ldots, J_k\}$

Correctness: Induction on # of jobs

BC: By definition, $OPT(\{\}) = 0$.

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 $OPT(J_1 \dots J_{j-1})$ & $OPT(I)$ (since $|I| < |J|$), we get correct value for $OPT(J_1 \dots J_j)$.

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
    option1 ← DP-Compute-Opt({J1,..., Jj-1})
    I ← set of jobs not overlapping Jj
    option2 ← w(Jj) + DP-Compute-OPT(I)
    M[j] ← max(option1, option2)
    return M[j]
```

full set of jobs

empty

Initial call will be DP-Compute-OPT(J, n, M)

Alternatively, could use a for loop.

Would fill in M starting at M[1].

At each stage, need to check M[j-1] and M[i]

↑ whatever i gives my set I

⚠ 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 + an addition + a max comparison
⇒ $O(1)$. (Actually $O(n)$ to compute I.)

Each entry only gets filled in once.

⇒ $\boxed{O(n^2)}$