CS 161: Design and Analysis of Algorithms

Dynamic Programming I: Weighted Interval Scheduling

- Example: Fibonacci Numbers
- Recurrence Trees
- Dynamic Programming Dags
- Weighted Interval Scheduling

Example: Fibonacci Numbers

- F(n) = {
  - 0 if n = 0
  - 1 if n = 1
  - F(n-1) + F(n-2) if n > 1

  }

Recursive Algorithm

- Fib1(n) = {
  - If n < 2, return n
  - Fib1(n-1) + Fib1(n-2) otherwise

  }

Recursive Algorithm

- Running Time?
  - Claim: For n>0, number of additions is F(n)-1
  - True for n = 1, 2
  - Inductively assume true for k < n
  - Fib1(n) uses 1 addition, plus the additions of Fib1(n-1) and Fib1(n-2)
  - Number of additions: 1 + (F(n-1)-1) + (F(n-2)+1)

Recursive Algorithm

- Running Time?
  - $\Omega(F(n))$
  - How fast does F(n) grow?

Fibonacci Growth Rate

- Let φ and ψ be solutions to $x^2 = x + 1$
  - $\varphi \approx 1.62$, the golden ratio
  - $\Psi \approx -0.62$
- Claim: $F(n) = \theta(\varphi^n)$
- Stronger Claim: $F(n) = (\varphi^n - \Psi^n)/(\varphi - \Psi)$
- Proof: True for $n = 0, 1$
  - Assume true for $k < n$
  - $F(n) = F(n-1) + F(n-2)$

Fibonacci Growth Rate

- F(n) = (φ^n – Ψ^n)/(φ – Ψ)
- Proof: True for n = 0, 1
  - Assume true for k < n

$$F(n) \quad = F(n-1) + F(n-2) = \frac{\varphi^{n-1} - \psi^{n-1}}{\varphi - \psi} + \frac{\varphi^{n-2} - \psi^{n-2}}{\varphi - \psi}$$

$$= \frac{\varphi^{n-2}(1+\varphi) + \psi^{n-2}(1+\psi)}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\varphi - \psi}$$

Recursive Algorithm

- Running Time?
  - $\Omega(F(n)) \approx \Omega(1.62^n)$
  - Grows extremely rapidly
  - Example: My computer
    - Running time ≈ 1.7 x $(1.62)^n$ nanoseconds
    - n = 30: 4 milliseconds
    - n = 40: 0.42 seconds
    - n = 50: 51 seconds
    - n = 60: 1.8 hours (projection)
    - n = 70: 9.1 days (projection)
    - n = 130: 93 billion years (projection)

Problem

- To compute F(n):
  - Compute F(n-1) and F(n-2)
  - Computing F(n-1) requires computing F(n-2) and F(n-3)
  - Computing F(n-2) requires computing F(n-3) and F(n-4)
  - …

Problem

- To compute F(n):
  - Call F(n-k) a total of F(k-1) times
  - Way too much repeated work

Solution: Memoization

- Remember answers to F(k) for future calls
- Keep track of (k,F(k)) mappings
  - Hash table
  - Array
- Fib2(n) = {
  - If(n < 2) return n
  - Check if Fib2(n) has been computed already, if so, output it
  - Otherwise, return Fib2(n-1) + Fib2(n-2)

  }

Alternative Approach

- We are going from top down
- How about going bottom up:
  - Keep array A, where A[k] = F(k)
  - Iteratively build array
    - First, set A[0] = 0, A[1] = 1
    - Then, for k = 2,…,n, set A[k] = A[k-1] + A[k-2]

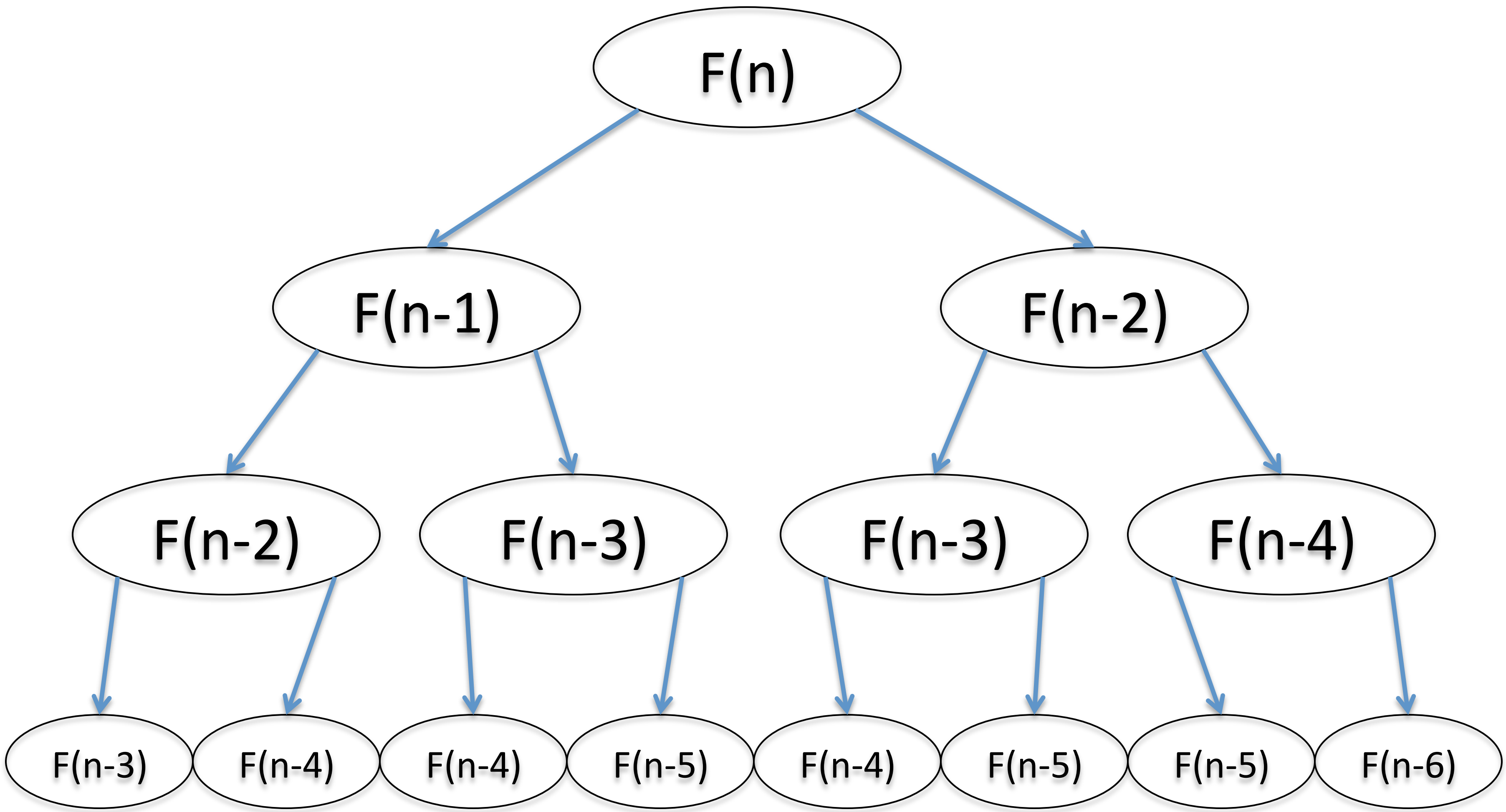
Iterative Algorithm

- Fib3(n) = {
  - Construct array A of length n+1
  - Set A[0] = 0, A[1] = 1
  - For k = 2, …, n
    - A[k] = A[k-1] + A[k-2]
  - Return A[n]

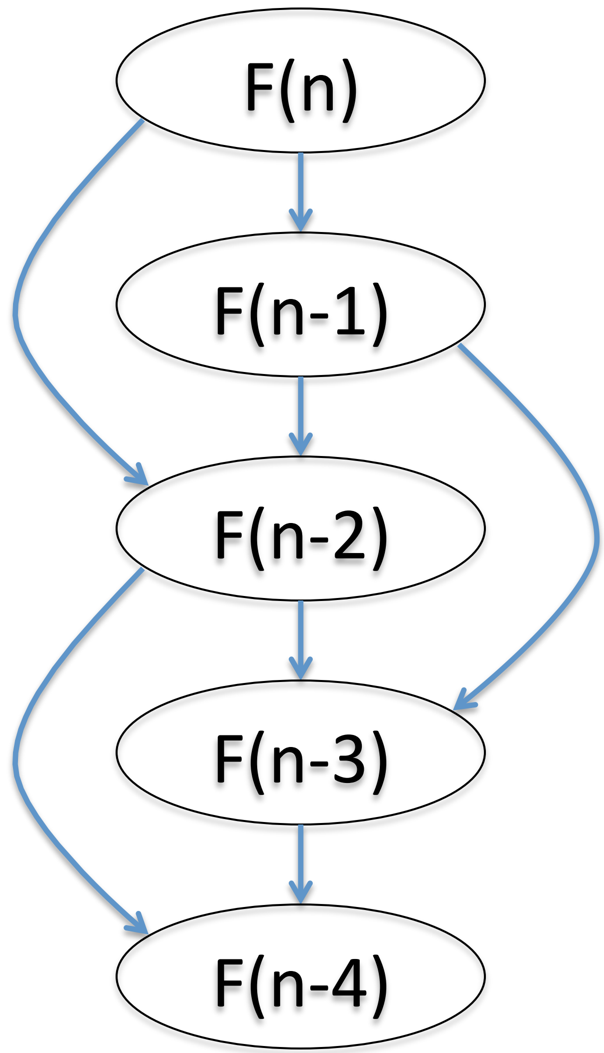
Iterative Algorithm

- Running Time:
  - n-1 iterations
  - Each step has a 1 addition (pretend all additions are constant time)
  - O(n) time
  - Much more tractable now

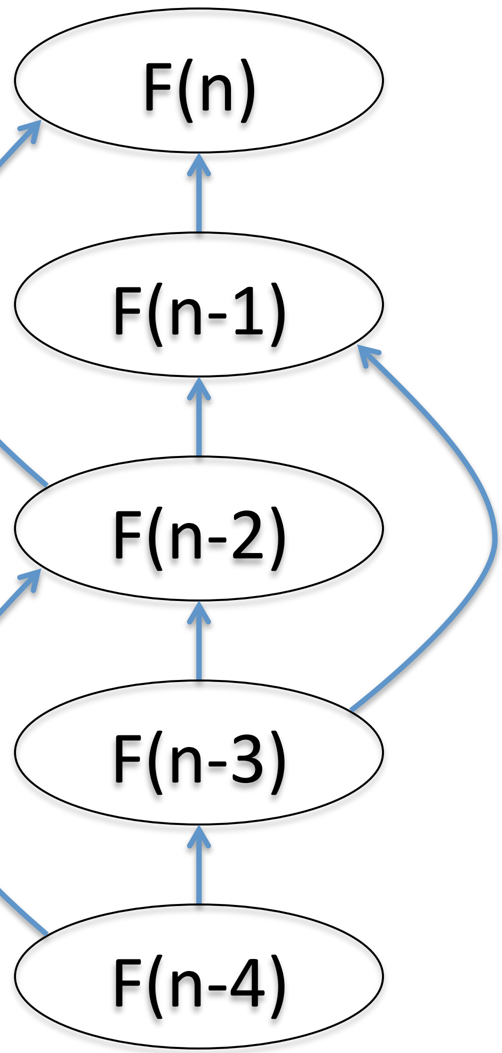
Recursion Tree



Merge Identical Nodes



Process Bottom Up



Dynamic Programming

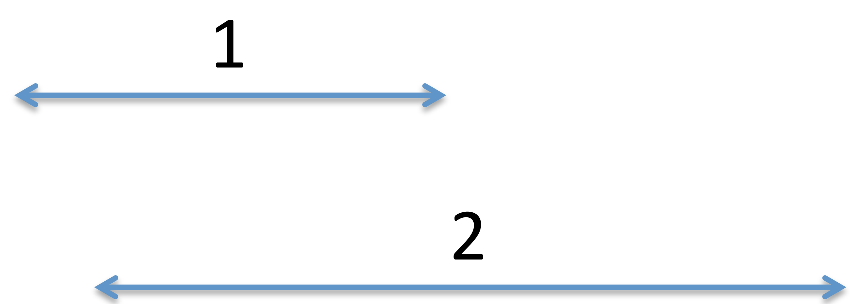
- Subproblems have dag structure
  - Edge represents prerequisite
- Solve subproblems in topological order
  - Whenever we solve a subproblem, we have already solved all of the other subproblems we need
- Very general, flexible tool

Weighted Interval Scheduling

- Given set of n intervals (s(i),f(i)), each with a weight w(i)
- Goal: pick set of overlapping intervals with largest possible weight
- If w(i) = 1, we have the unweighted scheduling problem, which can be solved by greedy
- Does greedy work here?

Weighted Interval Scheduling

- Recall greedy: pick interval that ends earliest



- This greedy approach does not work

Weighted Interval Scheduling

- Say we have optimal schedule S
- Two possibilities:
  - Interval n (the last one) is in S
  - Interval n is not in S

Weighted Interval Scheduling

- Suppose interval n is not in S
  - Then S is actually optimal for first n-1 intervals
  - Otherwise, any optimal solution for first n-1 intervals is solution for n intervals with higher weight

Weighted Interval Scheduling

- Suppose interval n is in S
  - Then no interval that overlaps n can be in S
  - S – {n} must be optimal over intervals that don't overlap interval n

Weighted Interval Scheduling

- Suggest the following approach:
  - Subproblems will consist of subsets of intervals
  - If subset has single interval, the optimal solution for that subproblem is just that interval
  - Otherwise, let T be some subset, and let t be the last interval
  - The optimal for a subset T is either:
    - The optimal for T – {t}
    - (The optimal for T – {s intersecting t}) + {t}

Problem

- There are 2^n subsets, so solving for all subsets will take exponential time
- Instead, we will be clever:
  - Order intervals by finish time (i.e. if $i < j$, $f(i) < f(j)$)
  - Let $p(i)$ be the last interval that ends before i starts, or 0 if no such interval
  - Now, we only need to solve problems on sets

    $\{1, 2, \ldots, k\}$

Weighted Interval Scheduling

- Optimal on {1, 2, ... , k}:
  - If interval k is not in optimal, then optimal is just optimal on {1, 2, …,k-1}
  - If k is in optimal, then optimal is the optimal on {1, 2, …, p(k)}, plus the interval k
  - Only need to check two cases

Weighted Interval Scheduling

- WeightedIntervalSchedule:
  - Create solution array S of length n+1
  - Create weight array W of length n+1
  - Sort intervals by f(i)
  - S[0] = {}, W[0] = 0, S[1] = {1}, W[1] = w(1)
  - For k = 2, …, n:
    - If W[k-1] < W[p(k)] + w(k), then:
      - W[k] = W[p(k)] + w(k)
      - S[k] = S[p(k)] + {k}
    - Else W[k] = W[k-1], S[k] = S[k-1]
  - Output S[n]

Running Time

- Say we have p(i) values, and list already sorted.
- Then, each iteration takes only O(1), so O(n) overall
- Computing p(j)?
  - Obvious algorithm: $O(n^2)$

Underlying Dag

- Nodes represent sets {1,…,k}
- Pointer from set {1,…,k-1} to {1,…,k} for all k
- Pointer form set {1,…,p(k)} to {1,…,k} for all k

Dynamic Programming Outline

- Find good subproblems
- Express solution to suproblem k in terms of solutions to other subproblems
  - Solution to subproblem k needs to be efficiently computable given solutions to other subproblems
- Solve subproblems in topological order

More on Fibonacci

- Dynamic solution isn't necessarily best
- Once we've computed F(k-1) and F(k-2), we no longer need F(k-3), F(k-4), …, F(0)
- Save space: only keep around last two computed values

More on Fibonacci

- Keep around F(k) and F(k-1)
- To update:

$$
\begin{aligned}
F(k) &= F(k-1) + F(k-2) \\
F(k-1) &= F(k-1)
\end{aligned}
$$

More on Fibonacci

- Keep around F(k) and F(k-1)
- To update:

$$\begin{pmatrix} F(k) \\ F(k-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(k-1) \\ F(k-2) \end{pmatrix}$$

More on Fibonacci

- Can we do better than O(n) additions?

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F(1) \\ F(0) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

How to Compute Powers

- Say we have a set X where we can multiply elements together
- How do we compute $x^n = x * x * x * \dots * x$?
  - Obvious solution: compute x^{n-1} recursively, and multiply by x
  - Requires n-1 multiplications in the set

How to Compute Powers

- What if we are computing x^4?
  - First compute $y = x^2$, then compute y^2
  - Only 2 multiplications
- What about x^8?
  - Compute $y = x^4$ as above, then compute y^2
  - Only 3 multiplications

How to Compute Powers

- In general, can compute x^{2^n} using n multiplications
- What about exponents that are not powers of 2?

How to Compute Powers

- Pow(x, n) = {
  - If n = 1, return x
  - If n is even, return Pow(x * x, n/2)
  - If n is odd, return x * Pow(x * x, (n-1)/2)

  }

How to Compute Powers

- Number of multiplications?
  - At most 2 per call to Pow
  - Exponent is at least divided by 2
  - O(log n) multiplications

More on Fibonacci

- Can we do better than O(n) additions?

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F(1) \\ F(0) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- Can compute using O(log n) 2x2 matrix multiplications
- O(log n) additions and multiplications

The Catch

- The integers we are adding and multiplying are large (exponential, in fact)
- Number of digits: O(n)
- Even though O(log n) additions and multiplications, each addition and multiplication takes time up to O(M(n)), where M(n) is the time to multiply 2 n-digit integers

Actual Running Time

- $T(n) = T(n/2) + O(M(n))$
- $M(n)$ is at least n, so running time dominated by $O(M(n))$ term
- Therefore, $T(n) = O(M(n))$