6.006	Lecture 19	Nov. 22, 2011

TODAY: Dynamic Programming I (of 4)

- memoization & subproblems: bottom up
- Fibonacci
- shortest paths
- guessing & DAG view

} examples

Dynamic programming: (DP) - big idea, hard, yet simple

powerful algorithmic design technique

- large class of seemingly exponential problems have a polynomial solution ("only") via DP
- particularly for optimization problems (min/max) (e.g. shortest paths)

* DP ≈ careful brute force

* DP ≈ recursion + "re-use"

History: Richard E. Bellman (1920-1984) → IEEE Medal of Honor, 1979

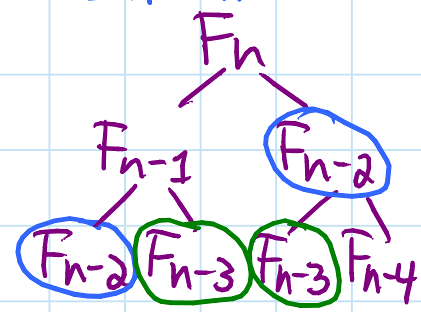
"Bellman... explained that he invented the name 'dynamic programming' to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who 'had a pathological fear and hatred of the term, research.' He settled on the term 'dynamic programming' because it would be difficult to give a 'pejorative meaning' and because 'It was something not even a Congressman could object to.'" [John Rust 2006]

Fibonacci numbers: $F_1 = F_2 = 1$; $F_n = F_{n-1} + F_{n-2}$

- goal: compute F_n

Naïve algorithm: follow recursive definition

```
fib(n):
    if n ≤ 2: f = 1
    else: f = fib(n-1) + fib(n-2)
    return f
```



$$\Rightarrow T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$$
$$\geq 2T(n-2) + O(1) \geq 2^{n/2}$$

EXPONENTIAL – BAD!

Memoized DP algorithm: remember, remember!

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    if n ≤ 2: f = 1
    else: f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

$\Rightarrow$ fib(k) only recurses first time called, $\forall k$

$\Rightarrow$ only n nonmemoized calls: $k = n, n-1, \ldots, 1$

- memoized calls free ($\Theta(1)$ time)

$\Rightarrow \Theta(1)$ time per call (ignoring recursion)

POLYNOMIAL – GOOD!

* DP $\approx$ recursion + memoization
  - memoize (remember) & re-use solutions to subproblems that help solve problem
    - in Fibonacci, subproblems are $F_1, F_2, \dots, F_n$

* $\Rightarrow$ time $= \underbrace{\#\text{ subproblems}} \cdot \underbrace{\text{time/subproblem}}$
  - Fibonacci: $n \cdot \Theta(1) = \Theta(n)$

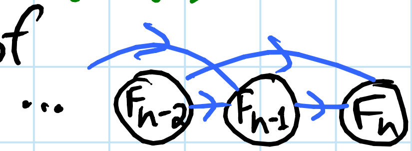
    ignore recursion!

Bottom-up DP algorithm:

```
fib = {}
for k in [1, 2, ..., n]:
    if k ≤ 2: f = 1
    else: f = fib[k-1] + fib[k-2]
    fib[k] = f
return fib[n]
```

$\Theta(1)$ (loop body), $\Theta(n)$ (loop)

- exactly the same computation as memoized DP (recursion "unrolled")
- in general: topological sort of subproblem dependency DAG

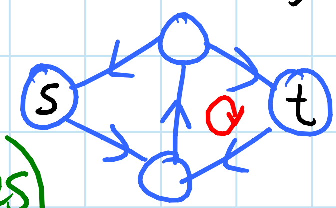


- practically faster: no recursion
- analysis more obvious
- can save space: just remember last 2 fibs $\Rightarrow \Theta(1)$

[side note: there is also an $O(\lg n)$-time algorithm for Fibonacci, via different techniques]

Shortest paths:

- recursive formulation:
  $$\delta(s,v) = \min\{\delta(s,u) + w(u,v) \mid (u,v) \in E\}$$
- memoized DP algorithm:
  takes infinite time if cycles!
  (kinda necessary to handle neg. cycles)



- works for directed acyclic graphs in $O(V+E)$
  ~ effectively DFS / topological sort + Bellman-Ford round rolled into a single recursion

* Subproblem dependency should be acyclic

- more subproblems remove cyclic dependence:
  $\delta_k(s,v)$ = shortest $s \to v$ path using $\leq k$ edges
- recurrence:
  $$\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$$
  $$\left.\begin{array}{l}\delta_\emptyset(s,v) = \infty \text{ for } s \neq v \\ \delta_k(s,s) = \emptyset \text{ for any } k\end{array}\right\} \text{base case}$$
  } if no neg. cycles
- goal: $\delta(s,v) = \delta_{|V|-1}(s,v)$

- memoize
- time: $\underbrace{\#\text{ subproblems}}_{v \leftarrow |V| \cdot |V| \rightarrow k} \cdot \underbrace{\text{time/subproblem}}_{O(V)} = O(V^3)$
  - actually $\Theta(\text{indegree}(v))$ for $\delta_k(s,v)$

  $\Rightarrow$ time $= \Theta\left(V \sum_{v \in V} \text{indegree}(v)\right) = \Theta(VE)$

  BELLMAN-FORD!

Guessing: how to design recurrence

- want shortest $s \to v$ path   (s) → ⋯ → (u) → (v)
- what is the last edge in path? dunno
- guess it's (u,v)

$\Rightarrow$ path is $\underbrace{\text{shortest } s \to u \text{ path}}_{\text{by optimal substructure}}$ + edge (u,v)

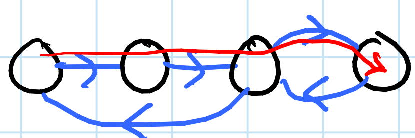
$\Rightarrow$ cost is $\underbrace{\delta_{k-1}(s,u)}_{\text{another subproblem}} + w(u,v)$

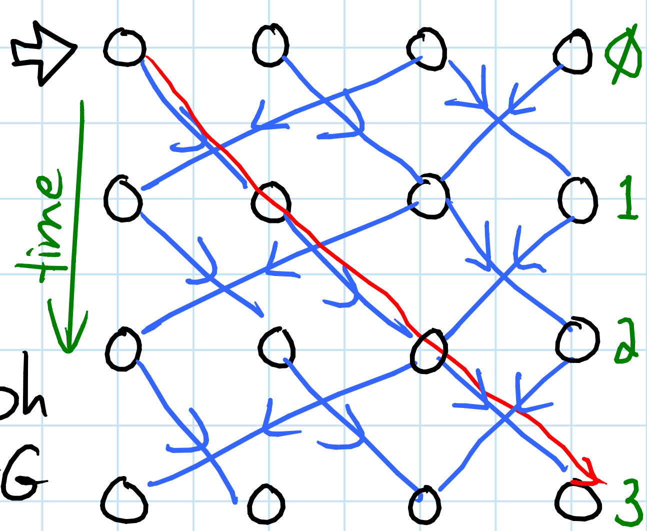
- to find best guess, try all & use best
  ↳ $|V|$ choices

* [– key: small (polynomial) # possible guesses per subproblem
  – typically this dominates time/subproblem

* [DP ≈ recursion + memoization + guessing

DAG view:





- like replicating graph to represent time
- converting shortest paths in graph → shortest paths in DAG

* [DP ≈ shortest paths in some DAG