Today Dynamic Programming I : "Zen and the art of Dynamic Programming"

- memoization & subproblems ; bottom up
- Examples:
  - Fibonacci
  - Shortest paths
- guessing & DAG view

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

powerful algorithmic design technique

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

* [DP ≈ careful brute force

* DP ≈ recursion + "re-use"]

<u>History</u>: 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 wold be difficult to give a 'pejorative meaning' and because 'It was something not even a Congressman could object to'"

[John Rust 2006]

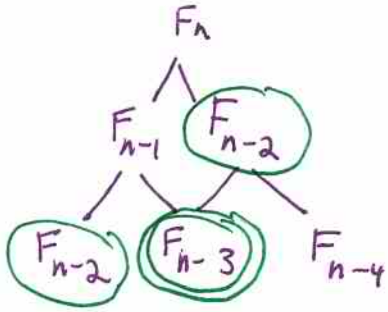
<u>Fibonacci numbers</u>:

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

<u>goal</u>: compute F_n

<u>Naive algorithm</u>: follow recursive definition

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



$\phi = 1.618\ldots$ golden ratio

<u>Runtime</u> $T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \phi^n$

$$\geq 2T(n-2) + O(1)$$
$$\geq 2^{n/2}$$

Exponential – BAD!

Problem: computing the subproblems over + over + over + over

<u>Memoized DP Algorithm</u>: 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 recurse <u>first</u> time called $\forall k$

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

- non memoized do $\Theta(1) + 2$ recursions
- memoized calls: only cost $\Theta(1)$ time total (no recursion)

total time: $\#\text{nonmemoized calls} * \Theta(1) + \underbrace{\#\text{memoized calls}}_{\leq 2 \cdot \#\text{nonmemoized calls}} \times \Theta(1)$

$$\leq n \times \Theta(1) + 2n \times \Theta(1)$$
$$= \Theta(n)$$

Polynomial – GOOD!

DP ≈ recursion + memoization

- memoize (remember) & reuse solutions to subproblems that help solve problem
  - in Fibonacci, subproblems are $F_1, F_2, \ldots, F_n$

$$\Rightarrow \text{time} = \underbrace{\#\text{ subproblems}}_{} \cdot \underbrace{\text{time/subproblem}}_{}$$

Fibonacci: $n \cdot \Theta(1) = \Theta(n)$

ignore recursion!

Bottom-up DP algorithm

```
fib = [none]*(n+1)
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]
```

$\Theta(1)$ $\Theta(n)$

reads array

- exactly same computation as memoized DP
  (recursion "unravelled")

- in general:

  topological sort of subproblem dependency DAG

$$
\begin{tikzcd}
\cdots \arrow[rr, bend left] & F_{n-2} \arrow[r] \arrow[rr, bend left] & F_{n-1} \arrow[r] & F_n
\end{tikzcd}
$$

- practically faster: avoids recursion
- analysis more obvious
- can save space: just remember last 2 fibs

  $\Rightarrow \Theta(1)$ space

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

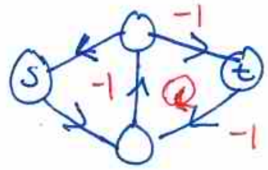
S.S. Shortest Paths:

A first try:

- recursive formulation:

$$\delta(s,v) = \min_u \{\delta(s,u) + w(u,v) \mid (u,v) \in E\}$$

- memoized DP algorithm:



Problem ⇒ takes <u>infinite</u> time if cycles!

(kind of necessary to handle negative cycles)

- works for directed acyclic graphs in $O(V+E)$

~ effectively DFS/topological sort + Bellman-Ford round rolled into a single recursion

What did we learn?

* [Subproblem dependency should be acyclic

- more subproblems remove cyclic dependence

$\delta_k(s,v)$ = shortest $s \to v$ path using $\leq k$ edges

- recurrence:

$$\delta_k(s,v) = \min_u \{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$$

$$\left.\begin{array}{ll} \delta_0(s,v) = \infty & \text{for } s \neq v \\ \delta_k(s,s) = 0 & \text{for any } k \end{array}\right\} \text{base case}$$

- goal: Find $\delta(s,v) = \delta_{|V|-1}(s,v)$ } if no neg. cycles

- memoize
- time: $\underbrace{\#\text{ subproblems}}_{|V|\cdot|V|}$ · time/subproblem

$|V|\cdot|V|$ · $O(V) = O(V^3)$

#possible u's → ← #possible K's   ↑ compute min of n u's

actually $\Theta(\text{indegree}(v))$ for $\delta_K(s,v)$

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

Bellman-Ford!

Guessing: how to design recurrence?

- want shortest $s \to v$ path (s) → .... → (u) → (v)
- what is last edge in path? dunno
- guess it's (u,v)!   this is good DP subproblem! same single source SP

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

by optimal substructure

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

another subproblem

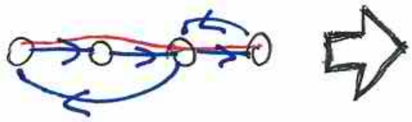
to find best guess, try all & use best
↳ $|V|$ choices

* [Key: small (polynomial) # guesses per subproblem

typically this dominates time/subproblem

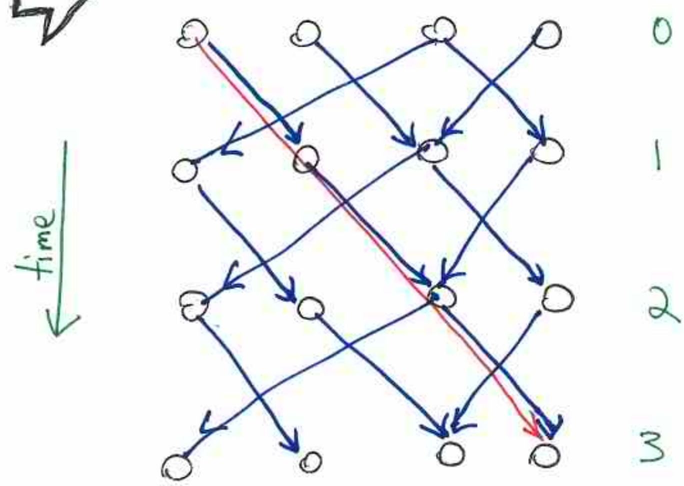
* [DP ≈ recursion + memoization + guessing

DAG View



- like replicating graph to represent time

- converting shortest paths in graph to shortest paths in DAG



* [DP often ≈ shortest paths in some DAG

↳ (but not always!)