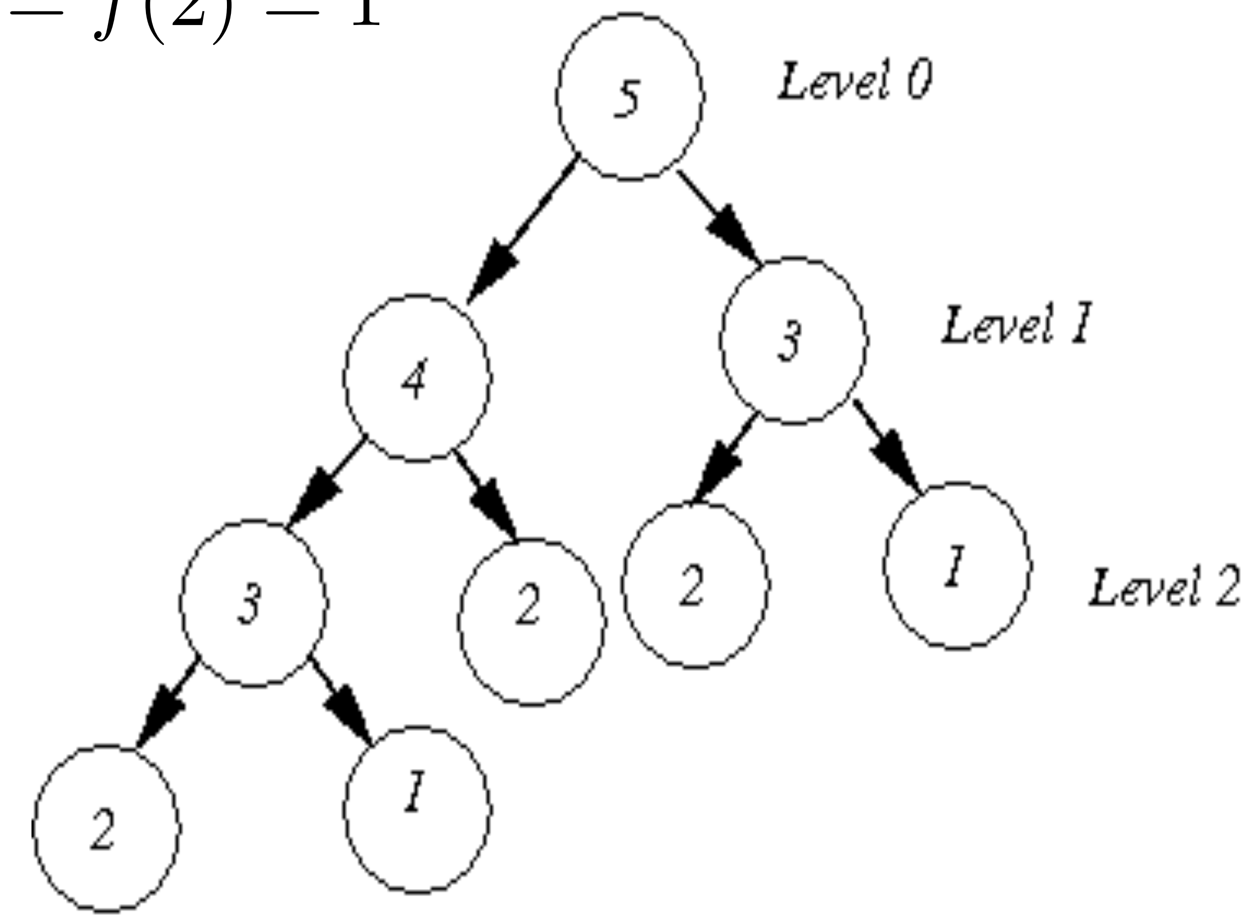
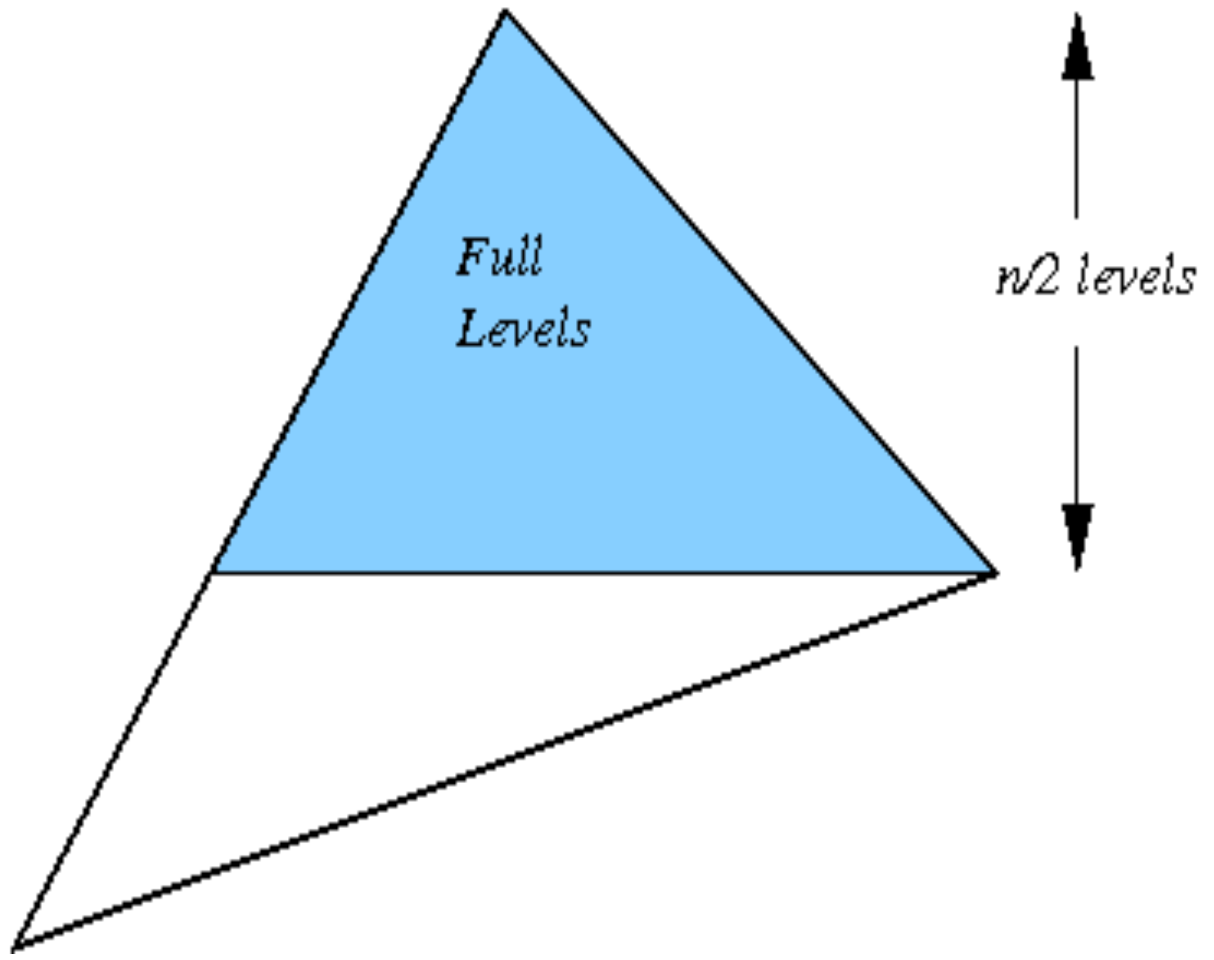
Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = f(2) = 1$$




```
def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)
```



naive recursion without memoization: $O(1.618...^n)$

DP2: bottom-up: $O(n)$

```
def fib0(n):
    a, b = 1, 1
    for i in range(3, n+1):
        a, b = a+b, a
    return a
```

DP1: top-down with memoization: $O(n)$

```
fibs={1:1, 2:1}
def fib1(n):
    if n not in fibs:
        fibs[n] = fib1(n-1) + fib1(n-2)
    return fibs[n]
```

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
  - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111
  - what about n=0?
  - first bit "1" followed by $f(n-1)$ substrings
  - first two bits "01" followed by $f(n-2)$ substrings

$$f(n) = f(n-1) + f(n-2)$$

$f(1)=2, f(0)=1$

Max Independent Set

- max weighted independent set on a linear-chain graph
  - e.g. 7 -- 2 -- 3 -- 5 -- 8
  - subproblem: $f(n)$ -- max independent set for a[1]..a[n]   (1-based index)

$$f(n) = \max\{f(n-1),\ \ f(n-2) + a[n]\}$$

$f(0)=0$; $f(1)=a[1]$?   better: $f(0)=0$; $f(-1)=0$

```
def max_wis2(a):
    best, back = {-1: 0, -2: 0}, {} # 0-based index!
    n = len(a)
    for i in range(n):
        best[i] = max(best[i-1], best[i-2]+a[i])
        back[i] = best[i] == best[i-1]
    return best[n-1], solution(n-1, a, back)

def solution(i, a, back):
    if i < 0:
        return []
    return solution(i-1, a, back) if back[i] else (solution(i-2, a, back) + [a[i]])
```

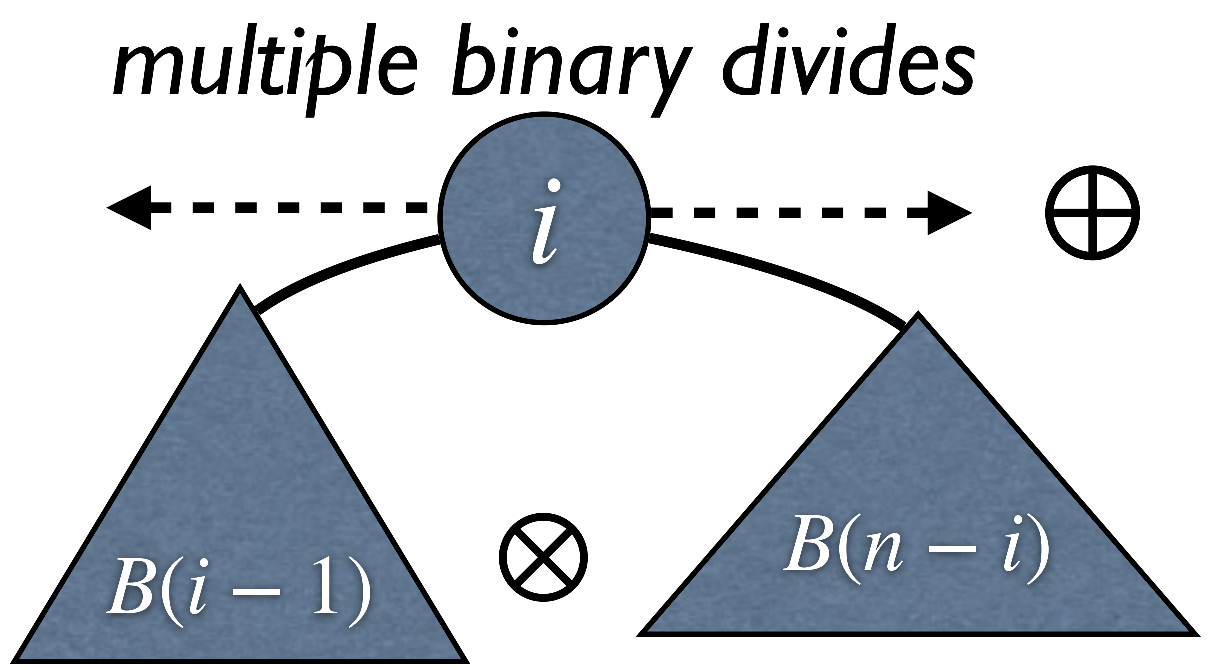
recursively backtrack the optimal solution

Summary

- Dynamic Programming = divide-n-conquer + overlapping subproblems
  - "distributivity" of work: $(a \otimes c) \oplus (b \otimes c) \oplus (a \otimes d) \oplus (b \otimes d) = (a \oplus b) \otimes (c \oplus d)$
- two implementation styles
  - 1. recursive top-down + memoization
  - 2. bottom-up
  - also need backtracking to recover best solution (recommended: backpointers)
- three steps in solving a DP problem
  - define the subproblem
  - recursive formula
  - base cases

Deeper Understanding of DP

- divide-n-conquer
  - single divide, independent conquer, combine
- DP = divide-n-conquer with multiple divides
  - for all possible divide
    - divide
    - conquer with memoization
    - combine subsolutions using the combination operator $\otimes$
  - summarize over all possible divides using summary operator $\oplus$
- multiple divides => overlapping subproblems
  - each single divide => independent subproblems!



	$\oplus$	$\otimes$
Fib	+	x
MIS	max	+
# BSTs	+	x
knapsack	max	+
shortest path	min	+

One-way vs. Two-way Divides

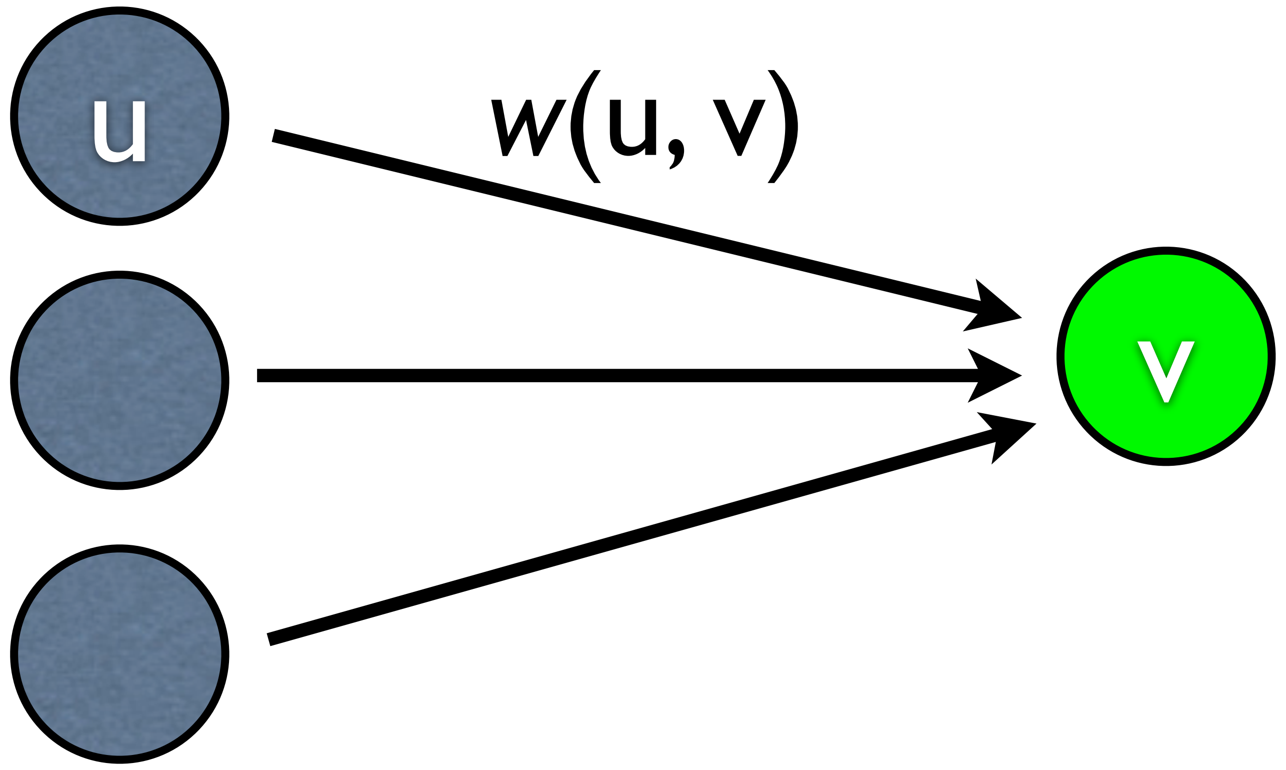
	two-way (binary divide)	one-way (unary divide)
divide-n-conquer	quicksort, best-case	quicksort, worst-case
	mergesort	quickselect
	tree traversal (DFS)	binary search
	heapify (top-down)	search in BST
DP	# of BSTs (hw5)	Fib, # of bitstrings (hw5)…
	optimal BST	max indep. set (hw5)
	RNA folding (hw10)	knapsack (all kinds, hw6)
	context-free parsing	Viterbi (hw8)
	matrix-chain multiplication, …	LCS, LIS, edit-distance,…

Two Divides vs. Multiple Divides (# of Choices)

	two divides	multiple divides
DP	Fib, # of bitstrings (hw5)…	# of BSTs (hw5)
	max indep. set (hw5)	unbounded knapsack (hw6)
	0-1 knapsack (hw6)	bounded knapsack (hw6)
		Viterbi (hw8)
		RNA folding (hw10)

Viterbi Algorithm for DAGs

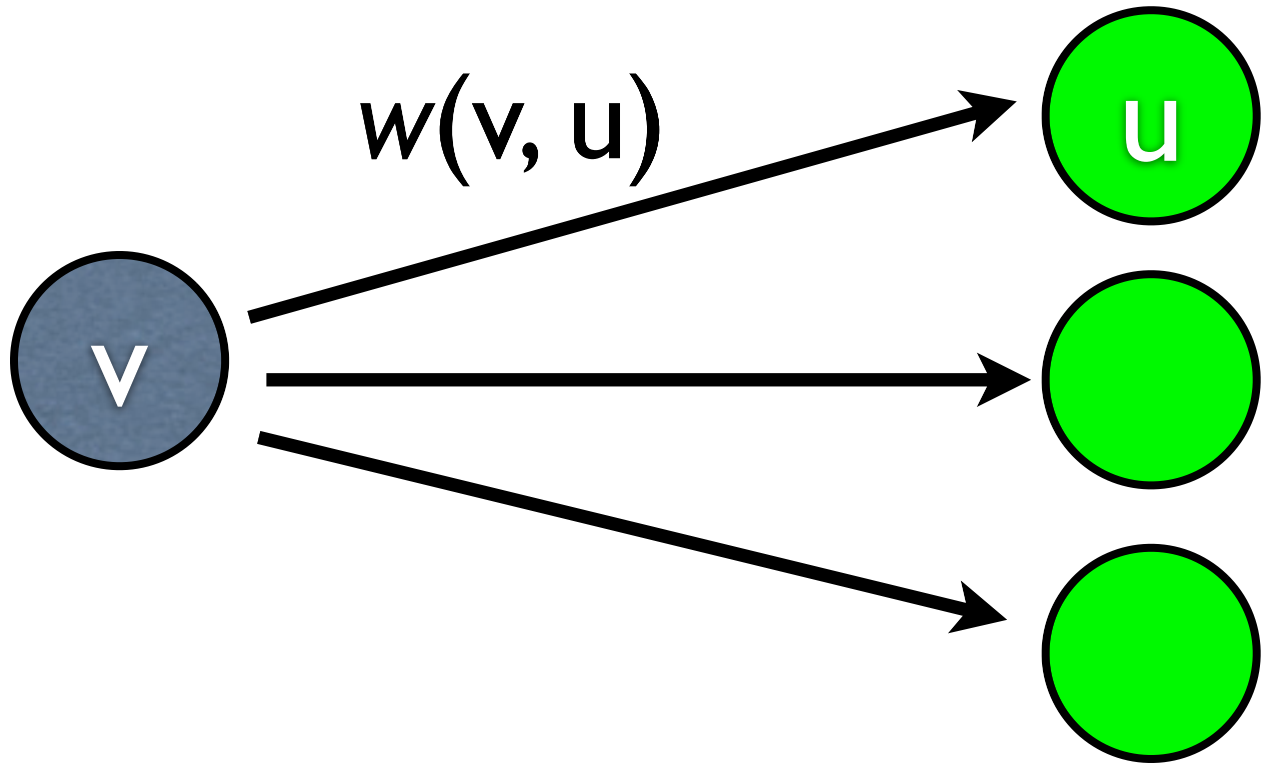
1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each incoming edge (u, v) in E
   - use d(u) to update d(v): $d(v) \oplus = d(u) \otimes w(u, v)$
   - key observation: d(u) is fixed to optimal at this time



- time complexity: O(V + E)

Variant 1: forward-update

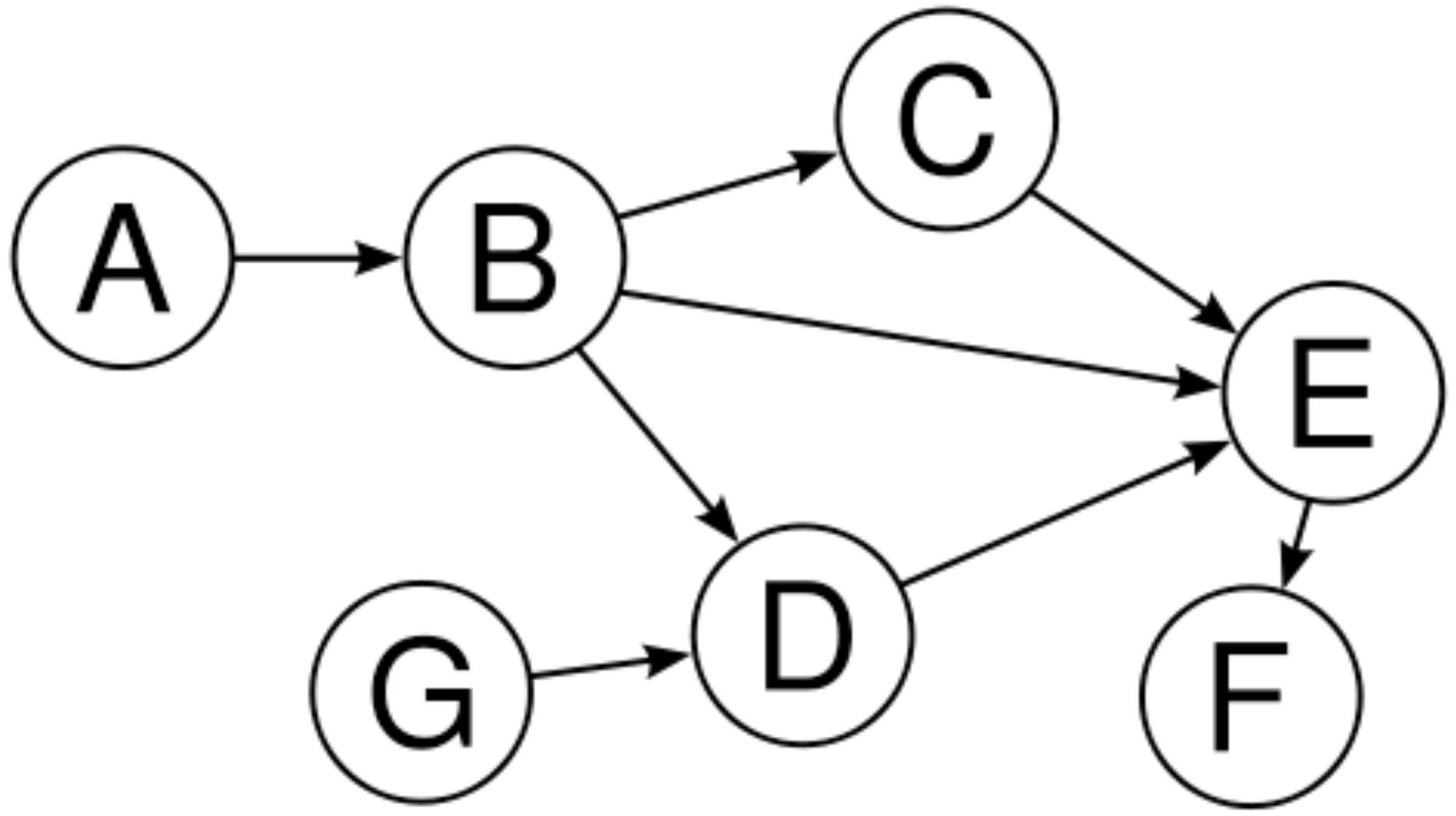
1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each outgoing edge (v, u) in E
   - use d(v) to update d(u): $d(u) \oplus = d(v) \otimes w(v, u)$
   - key observation: d(v) is fixed to optimal at this time



- time complexity: O(V + E)

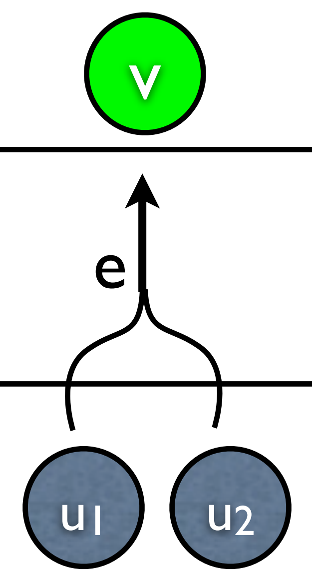
Variant 2: Recursive Descent

- Top-down Recursion + Memoization = Bottom-up
- Start from the target vertex, going backwards
  - remember each visited vertex
- Sometimes easier to implement
- There is a tradeoff b/w top-down and bottom-up



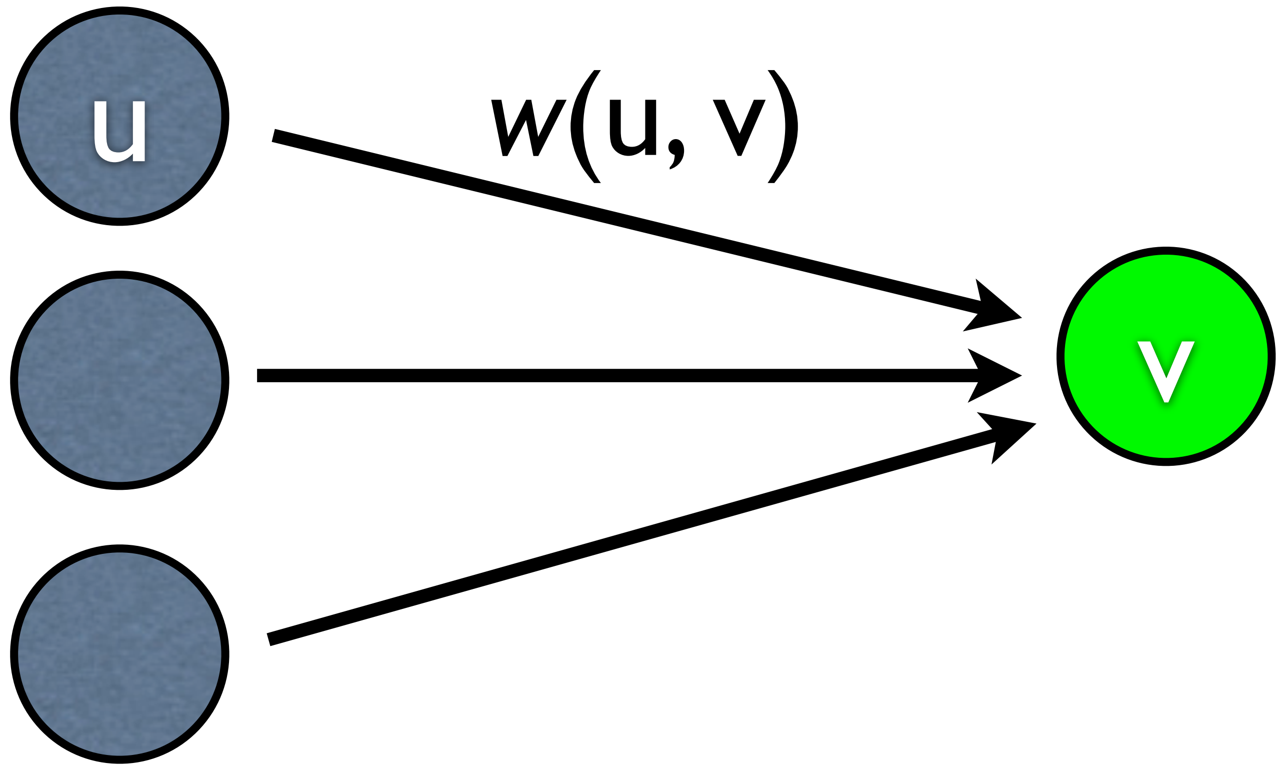
One-way vs. Two-way Divides (Graph vs. Hypergraph)

		two-way (binary divide)		one-way (unary divide)
divide-n-conquer	binary tree	quicksort, best-case	linear-chain	quicksort, worst-case
		mergesort		quickselect
		tree traversal (DFS)		binary search
		heapify (top-down)		search in BST
DP	hypergraph	# of BSTs (hw5)	graph	Fib, # of bitstrings (hw5)...
		optimal BST		max indep. set (hw5)
		RNA folding (hw10)		knapsack (all kinds, hw6)
		context-free parsing		Viterbi (hw8)
		matrix-chain multiplication, ...		LCS, LIS, edit-distance,...



Viterbi Algorithm for DAGs

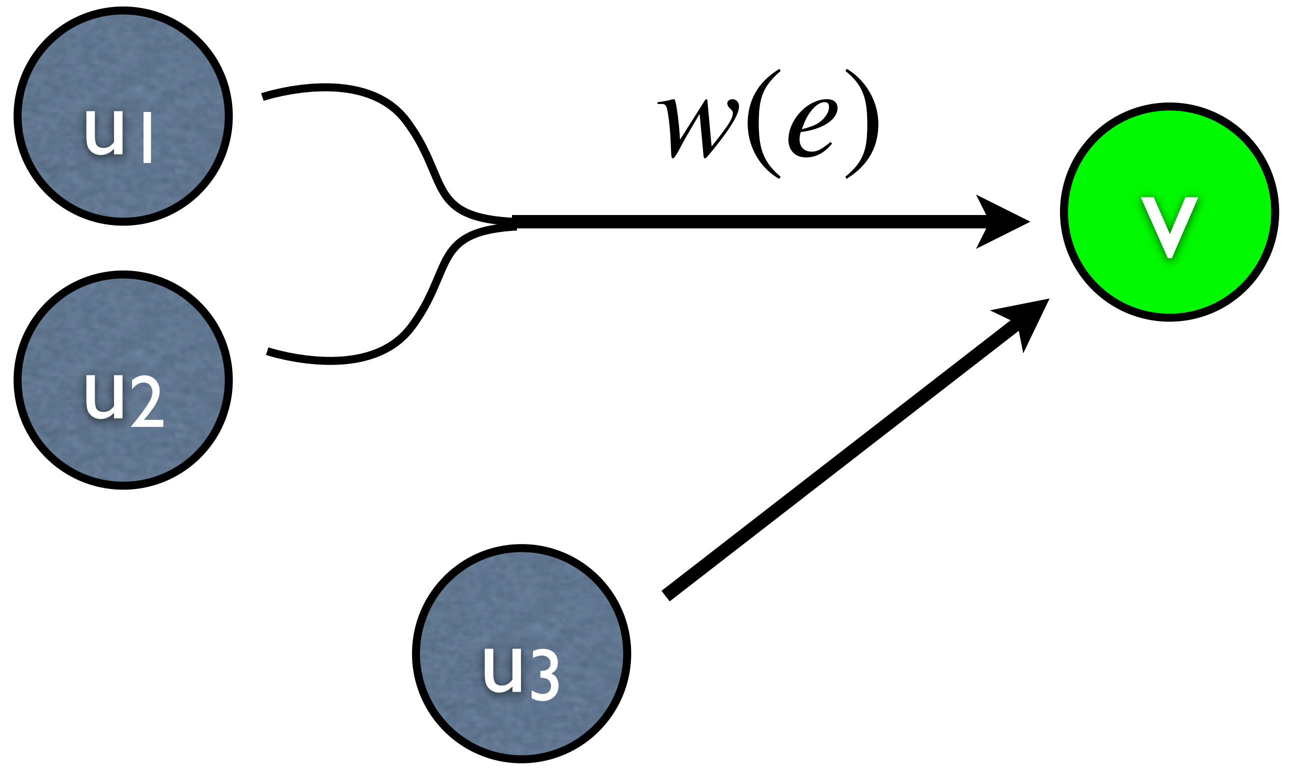
1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each incoming edge (u, v) in E
   - use d(u) to update d(v): $d(v) \oplus = d(u) \otimes w(u, v)$
   - key observation: d(u) is fixed to optimal at this time



- time complexity: O(V + E)

Viterbi Algorithm for DAHs

1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each incoming hyperedge $e = ((u_1, .., u_{|e|}), v, w(e))$
   - use $d(u_i)$'s to update $d(v)$
   - key observation: $d(u_i)$'s are fixed to optimal at this time

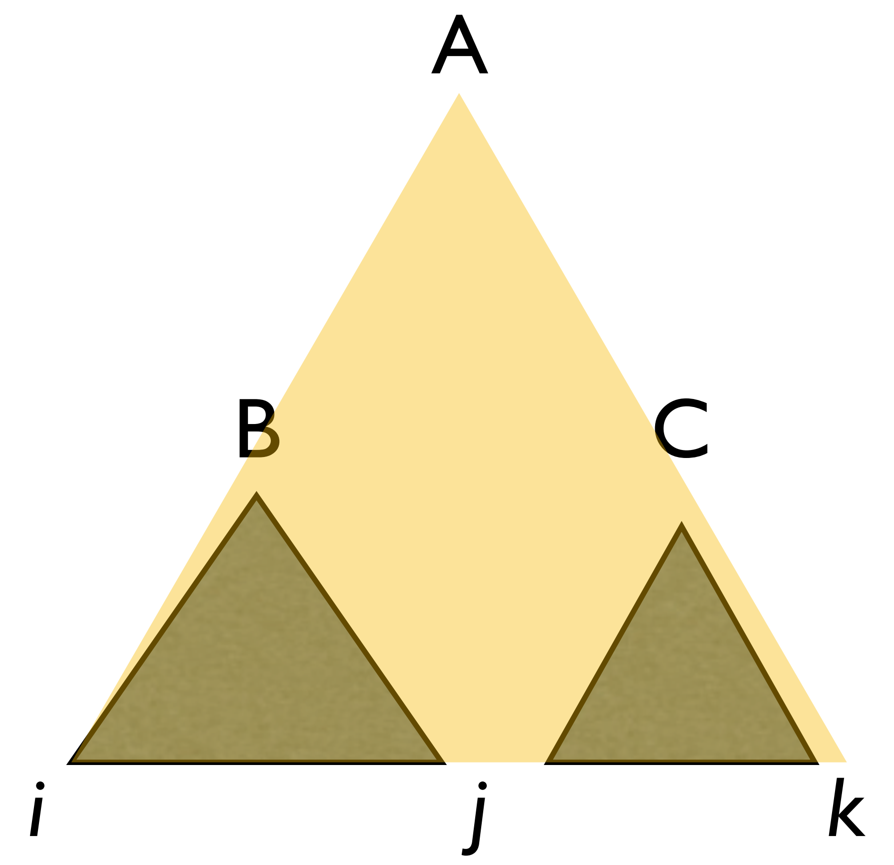


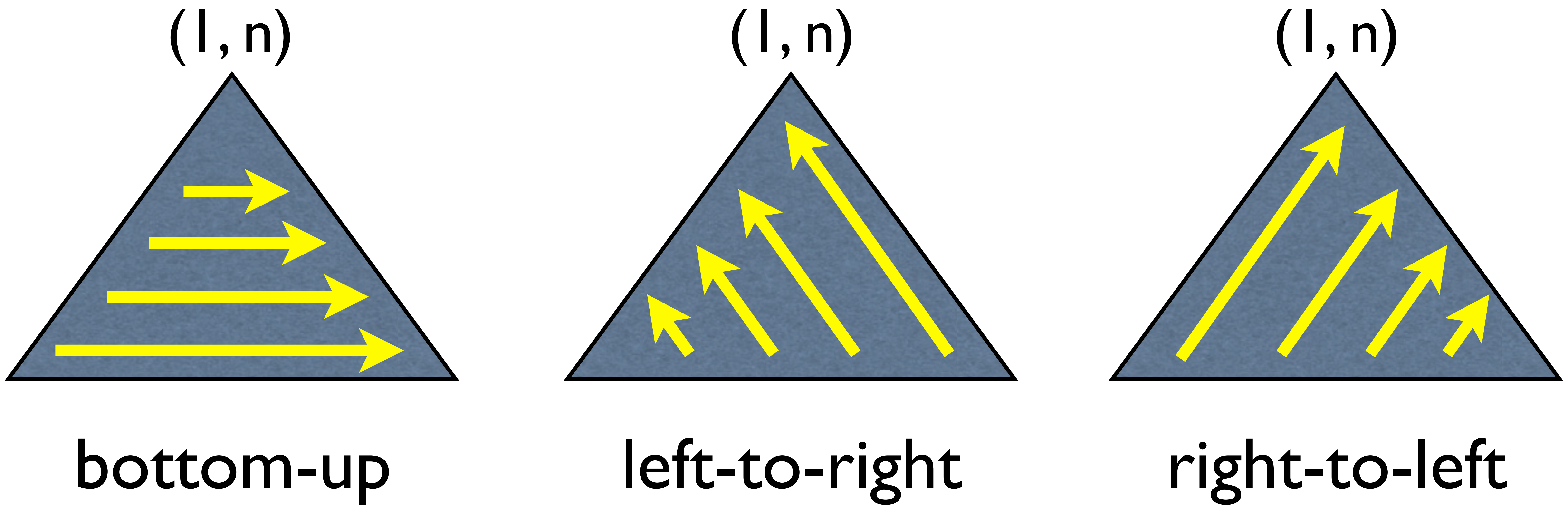
$$d(v) \oplus = d(u_1) \otimes d(u_2) \otimes w(e)$$

- time complexity: $O(V + E)$ (assuming constant arity)

Example: RNA Folding and CKY Parsing

- typical instance of the generalized Viterbi for DAHs
- many variants of CKY ~ various topological ordering
- Nussinov algorithm in RNA is almost identical to CKY but w/o overcounting





all $O(n^3)$

k-best Viterbi

- simple extension of Viterbi to solve k-best on graphs and hyper graphs