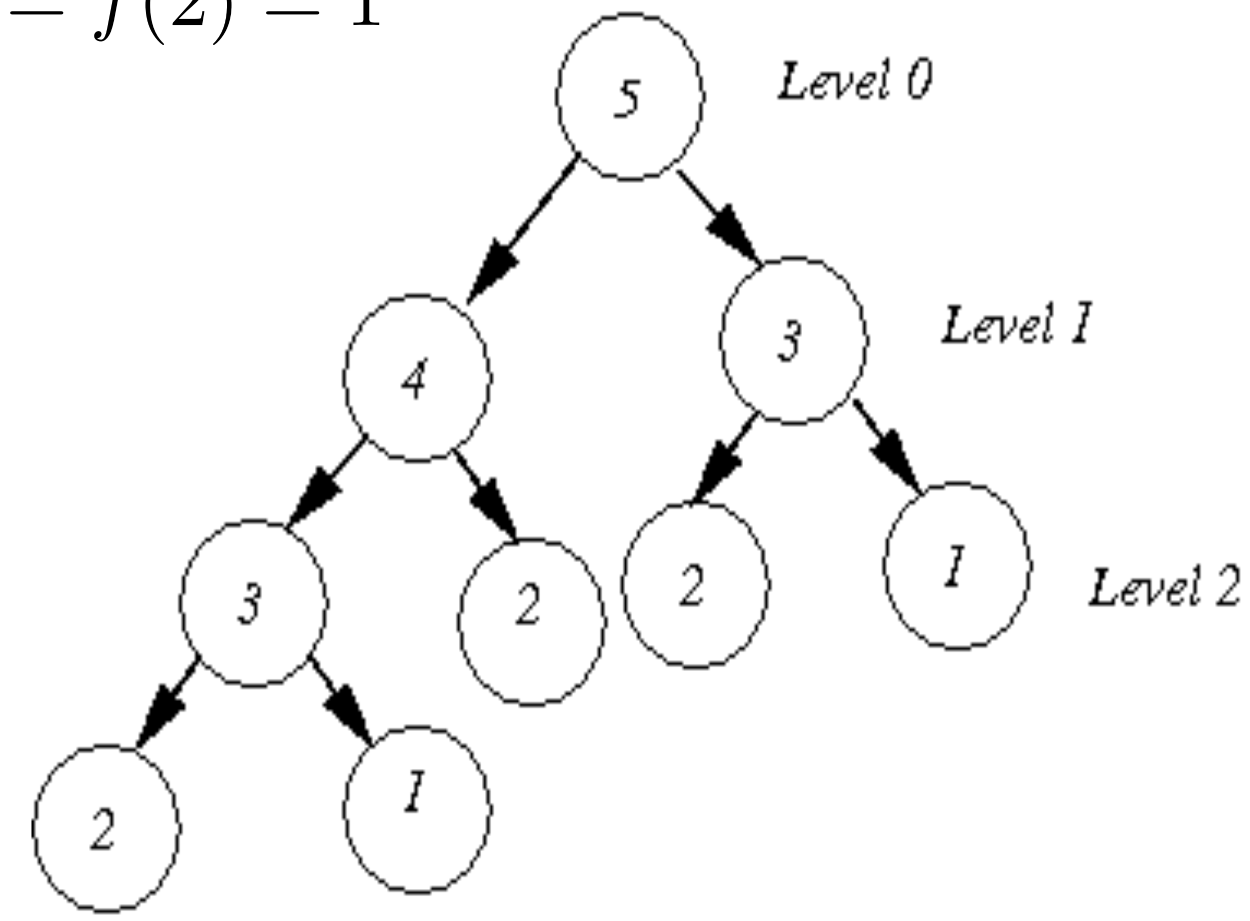
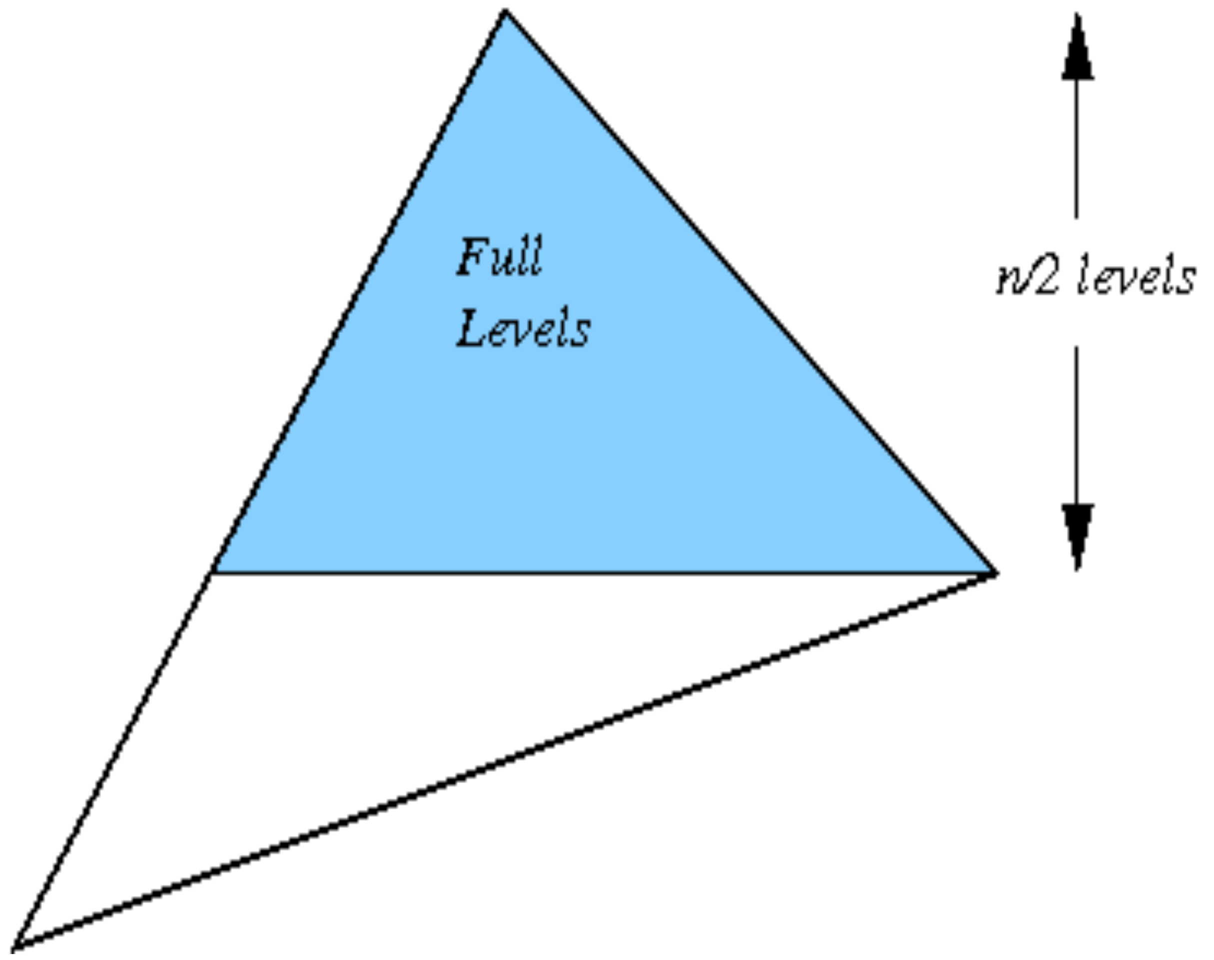
Dynamic Programming 101

- 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
```

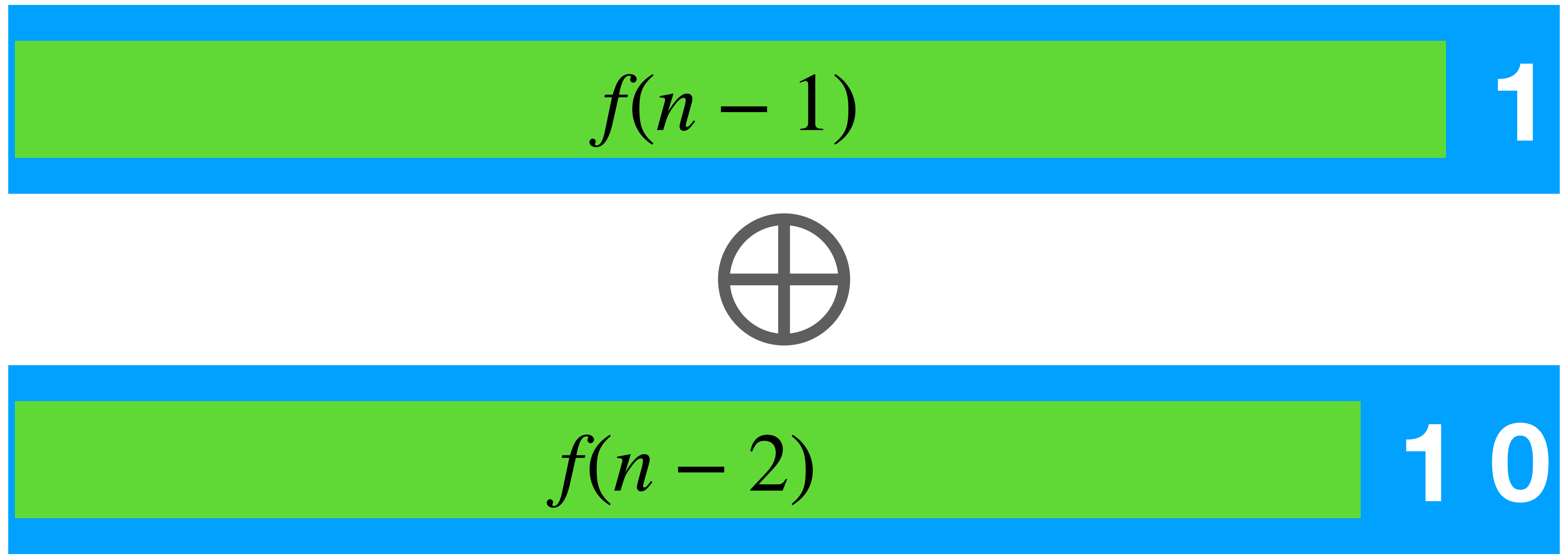
```
def fib0(n):
  f = [1, 1]
  for i in range(3, n+1):
    fibs.append(f[-1]+f[-2])
  return f[-1]
```

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

```
fibs={1:1, 2:1} # hash table (dict)
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?
  - last bit "1" followed by $f(n-1)$ substrings
  - last 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 (MIS)

- max weighted independent set on a linear-chain graph
  - e.g. **9** — 10 — **8** — 5 — 2 — **4** ; best MIS: [9, 8, 4] = 21 (vs. greedy: [10, 5, 4] = 19)
  - 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]? No! f(1)=max(a[1], 0)

or even better: *f*(0)=0; *f*(-1)=0

cost reward

MIS $f(n) = \max \begin{cases} f(n-1) + 0 \\ f(n-2) + a[n] \end{cases}$

bitstrings $f(n) = + \begin{cases} f(n-1) \times 1 \\ f(n-2) \times 1 \end{cases}$

summary operator ⊕ (across divides)

combination operator ⊗ (within a divide)

recursively backtrack the optimal solution

Summary

- Divide-and-Conquer = divide + conquer + combine
- Dynamic Programming = multiple divides + memoized conquer + summarized combine
- two implementation styles
  - 1. recursive top-down + memoization
  - 2. bottom-up
- backtracking to recover best solution for optimization problems
  - 1. backpointers (recommended); 2. store subsolutions (not recommended — often slows down); 3. recompute on-the-fly
- two operators: $\oplus$ for summary (across multiple divides) and $\otimes$ for combine (within a divide)
- counting problems vs. optimization problems ("cost-reward model")
- three steps in solving a DP problem
  - define the subproblem
  - recursive formula
  - base cases

cost   reward

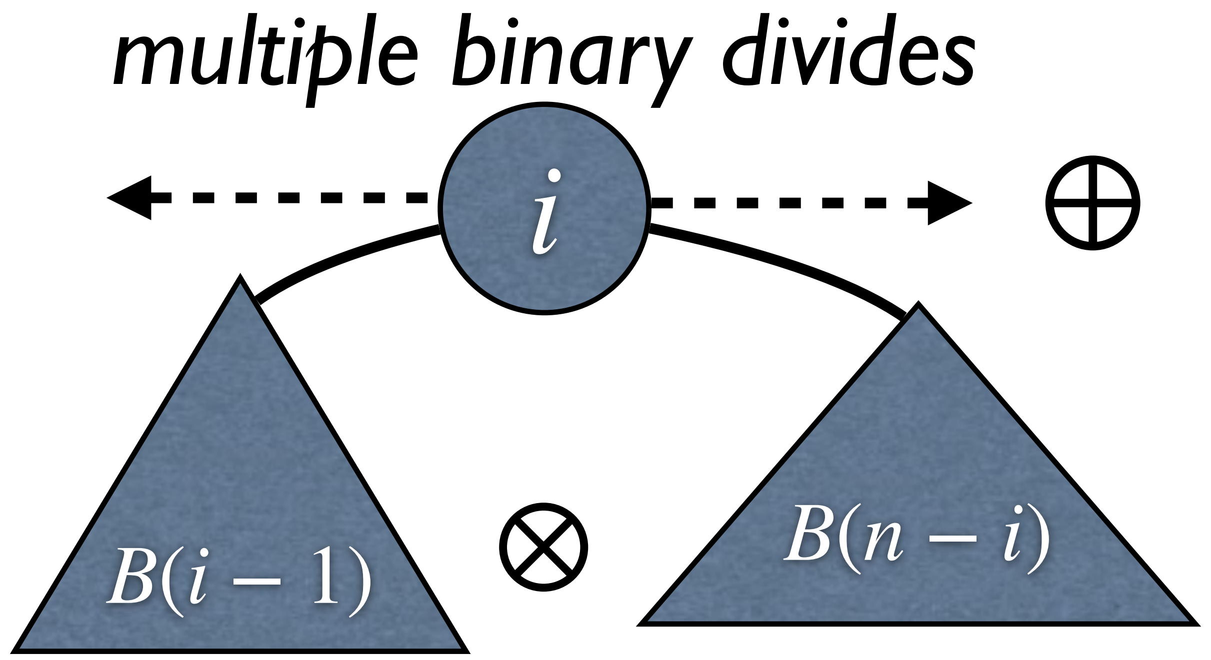
$$f(n) = \max \begin{cases} f(n-1) + 0 \\ f(n-2) + a[n] \end{cases}$$

summary operator $\oplus$ (across divides)

combination operator $\otimes$ (within a divide)

Deeper Understanding of DP

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

$$B(n) = \oplus_{i=1}^{n} \left(B(i-1) \otimes B(n-i) \right)$$

$$B(0) = 1$$

Unary vs. Binary Divides

$(a) : T(n) = 2T(n/2) + \ldots$

$(b) : T(n) = T(n-1) + \ldots$

$(c) : T(n) = T(n/2) + \ldots$

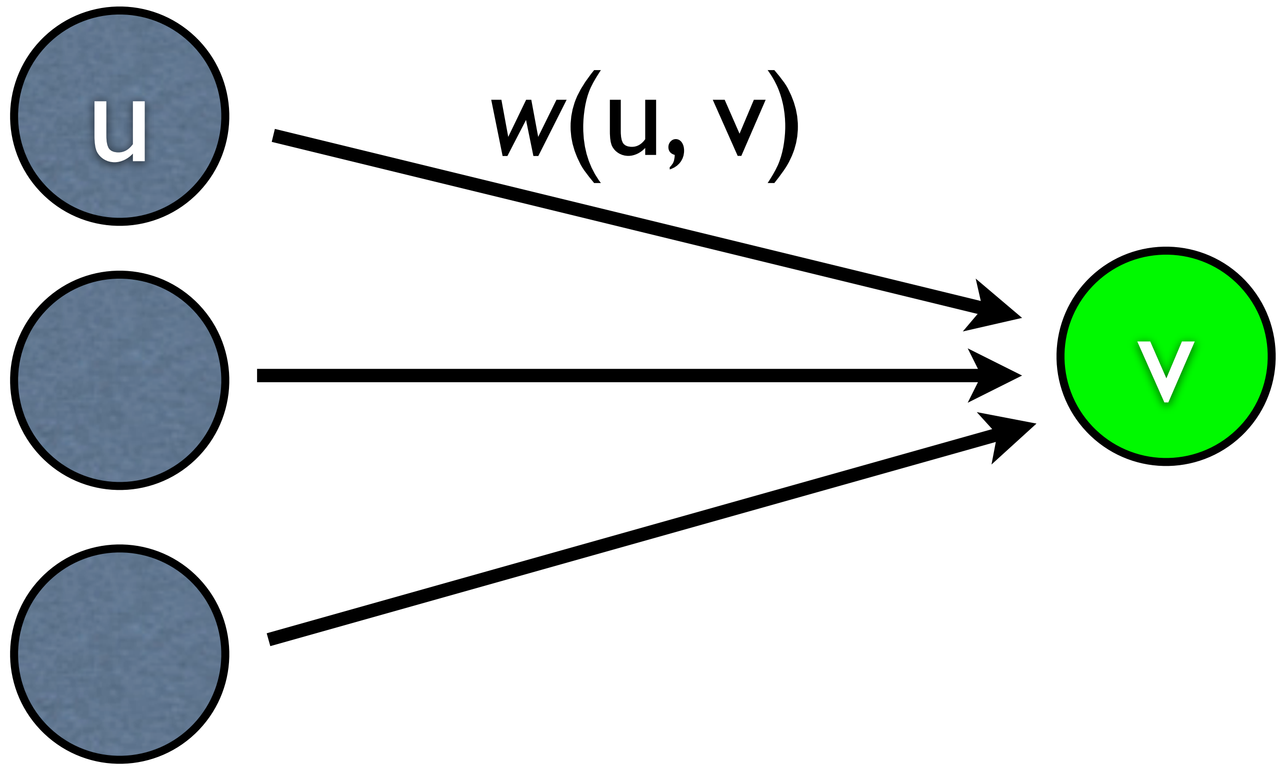
	branching (binary divide)	one-sided (unary divide)
divide-n-conquer	quicksort, best-case	quicksort, worst-case (*b*)
	mergesort	quickselect: worst (*b*), best (*c*)
	(balanced) tree traversal (DFS)	binary search: (*c*)
	heapify (top-down)	search in BST: worst (*b*), best (*c*)
DP	# of BSTs (hw5), *midterm*	Fib, # of bitstrings (hw5)…
	optimal BST, *final*	max indep. set (hw5)
	RNA folding (hw10)	knapsack (hw6), *midterm*
	context-free parsing	Viterbi (hw8), *final*
	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

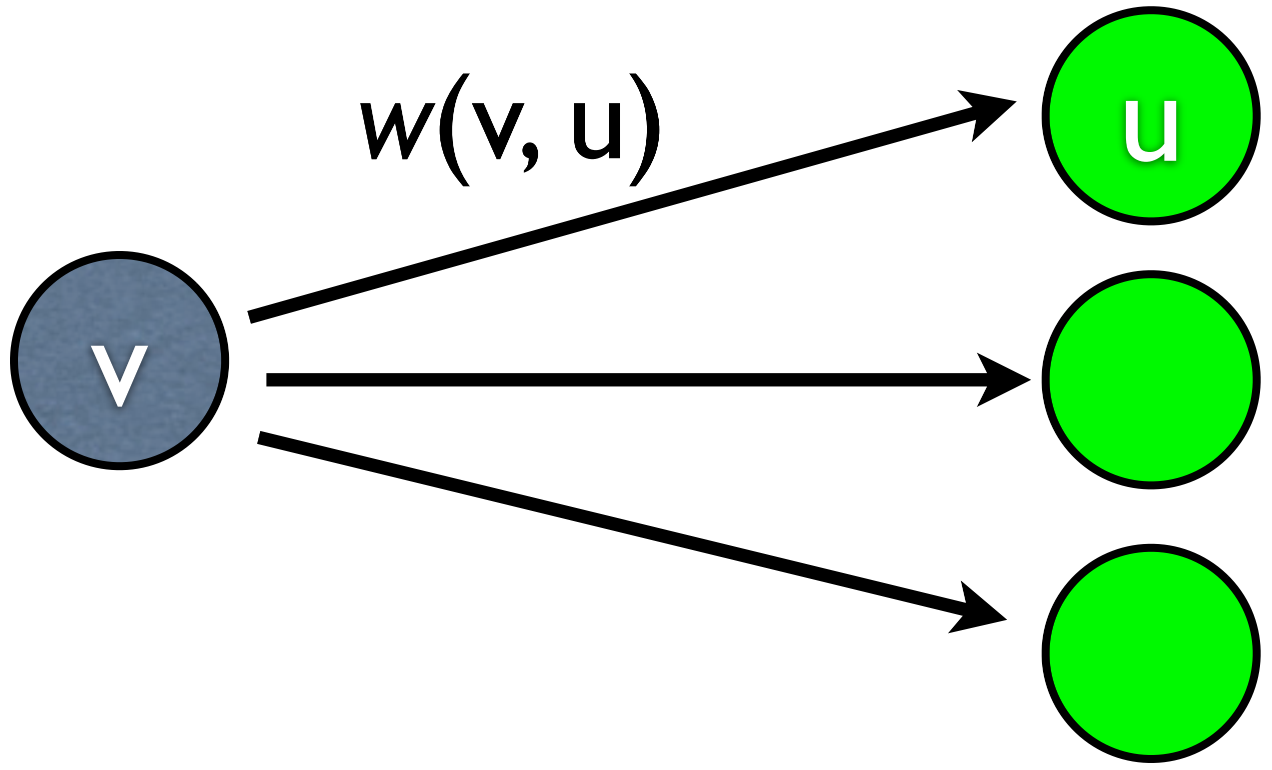
1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each incoming edge (u, v) in E
   - use d(u) to update d(v): $d(v) \oplus= d(u) \otimes w(u,v)$
   - key observation: d(u) is fixed to optimal at this time



- time complexity: O(V + E)

Variant 1: forward-update

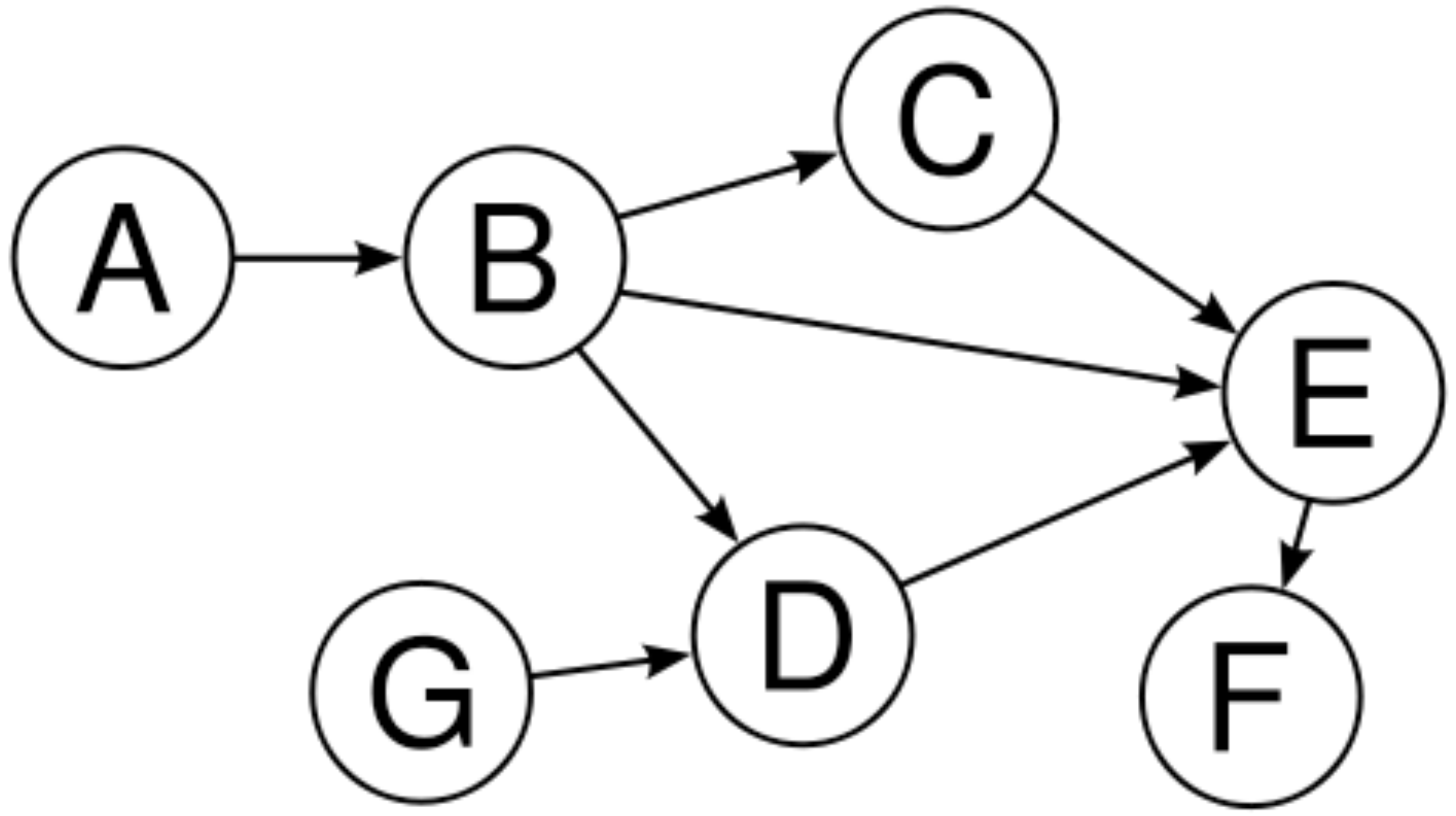
1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each outgoing edge (v, u) in E
   - use d(v) to update d(u): $d(u) \oplus = d(v) \otimes w(v, u)$
   - key observation: d(v) is fixed to optimal at this time



- time complexity: O(V + E)

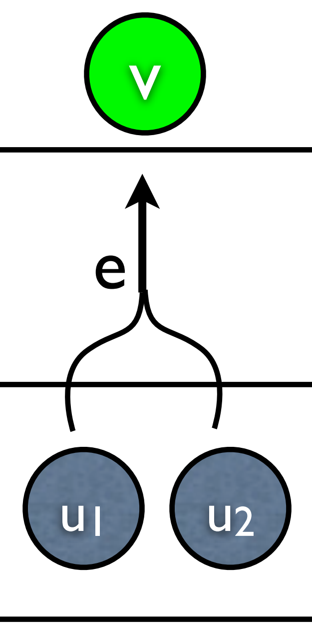
Variant 2: Recursive Descent

- 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



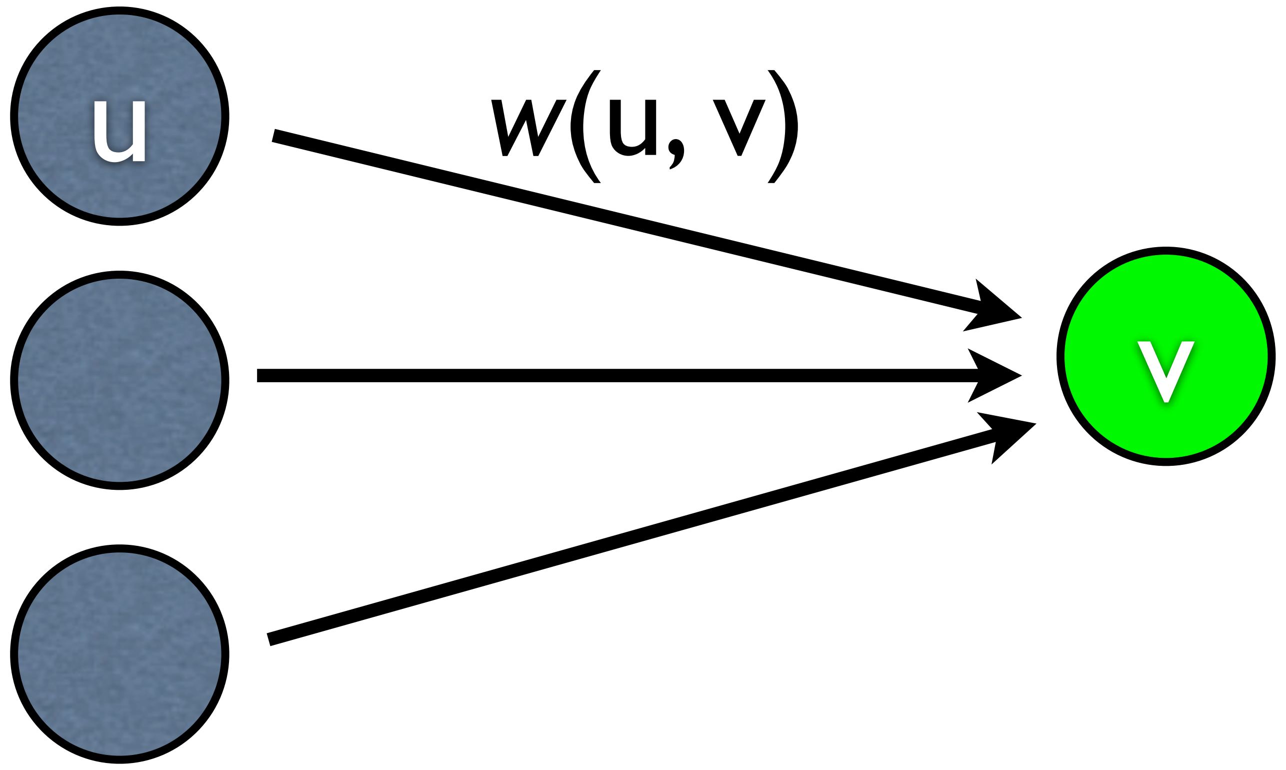
One-way vs. Two-way Divides (Graph vs. Hypergraph)

	two-way (binary divide)	one-way (unary divide)
divide-n-conquer	quicksort, best-case	quicksort, worst-case
	mergesort	quickselect
	tree traversal (DFS)	binary search
	heapify (top-down)	search in BST
	binary tree	*linear-chain*
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,...
	hypergraph	*graph*



Viterbi Algorithm for DAGs

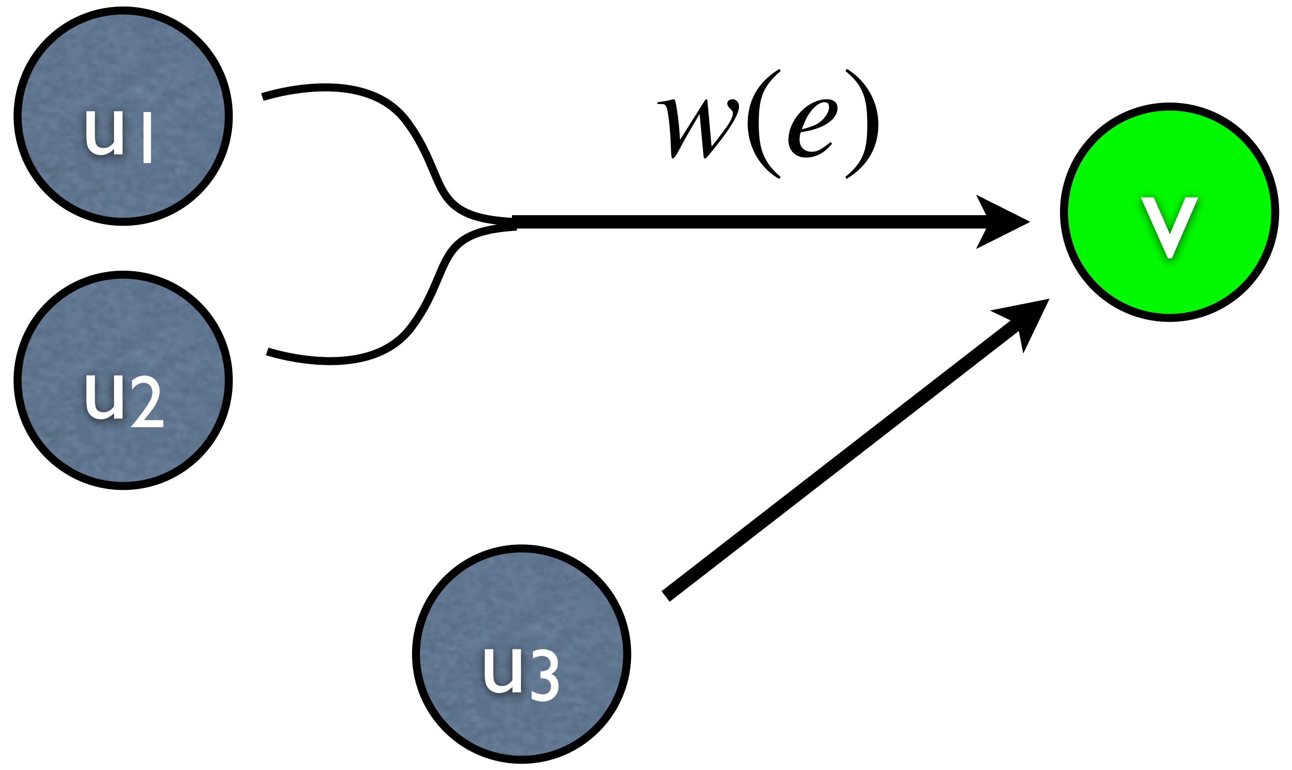
1. topological sort
2. visit each vertex v in sorted order and do updates
   - for each incoming edge (u, v) in E
   - use d(u) to update d(v): $d(v) \oplus = d(u) \otimes w(u, v)$
   - key observation: d(u) is fixed to optimal at this time



- 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

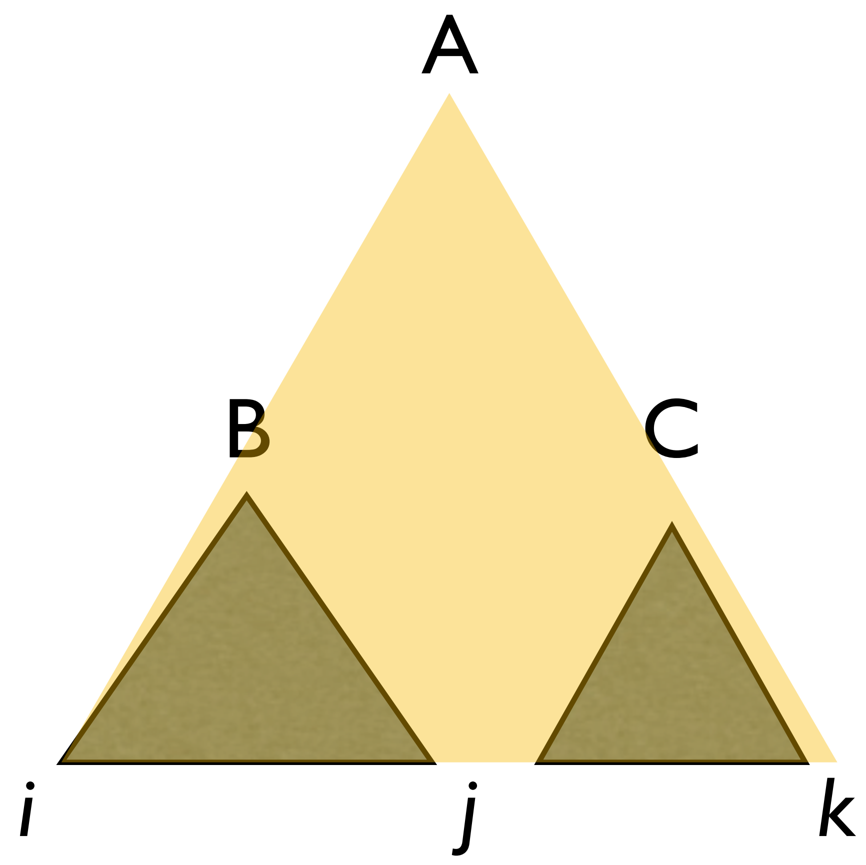


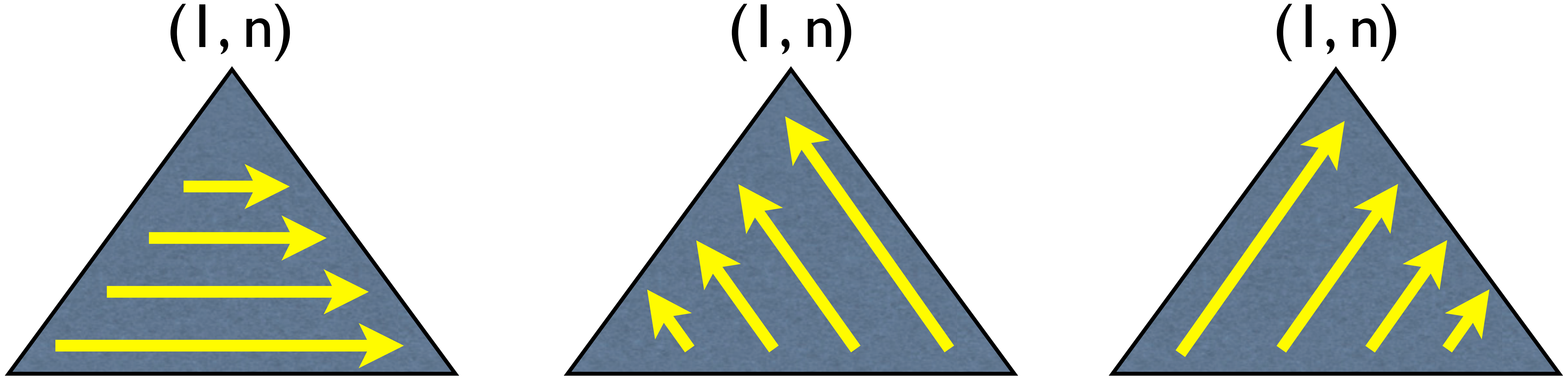
$$d(v) \oplus = d(u_1) \otimes d(u_2) \otimes w(e)$$

- 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

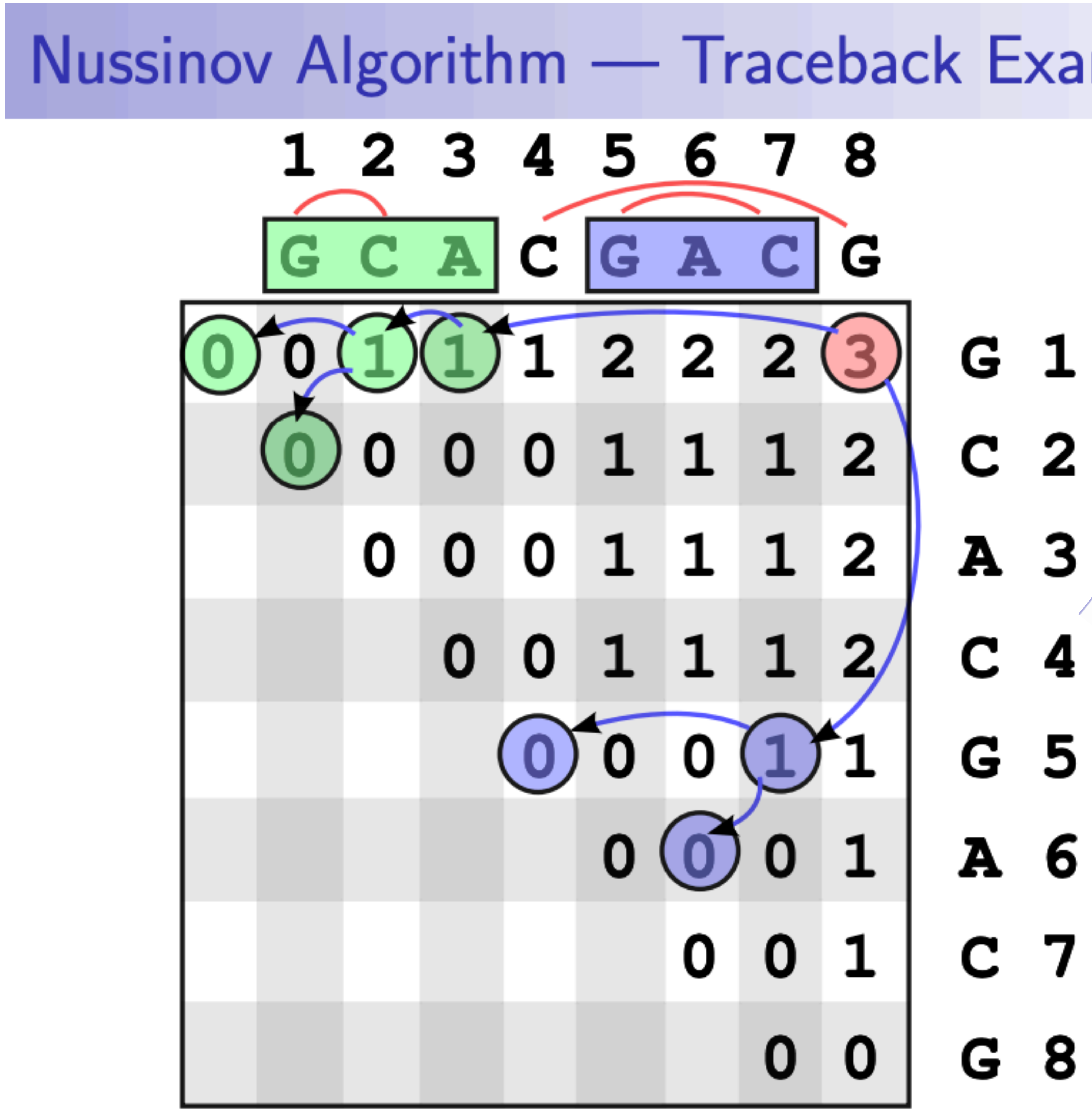


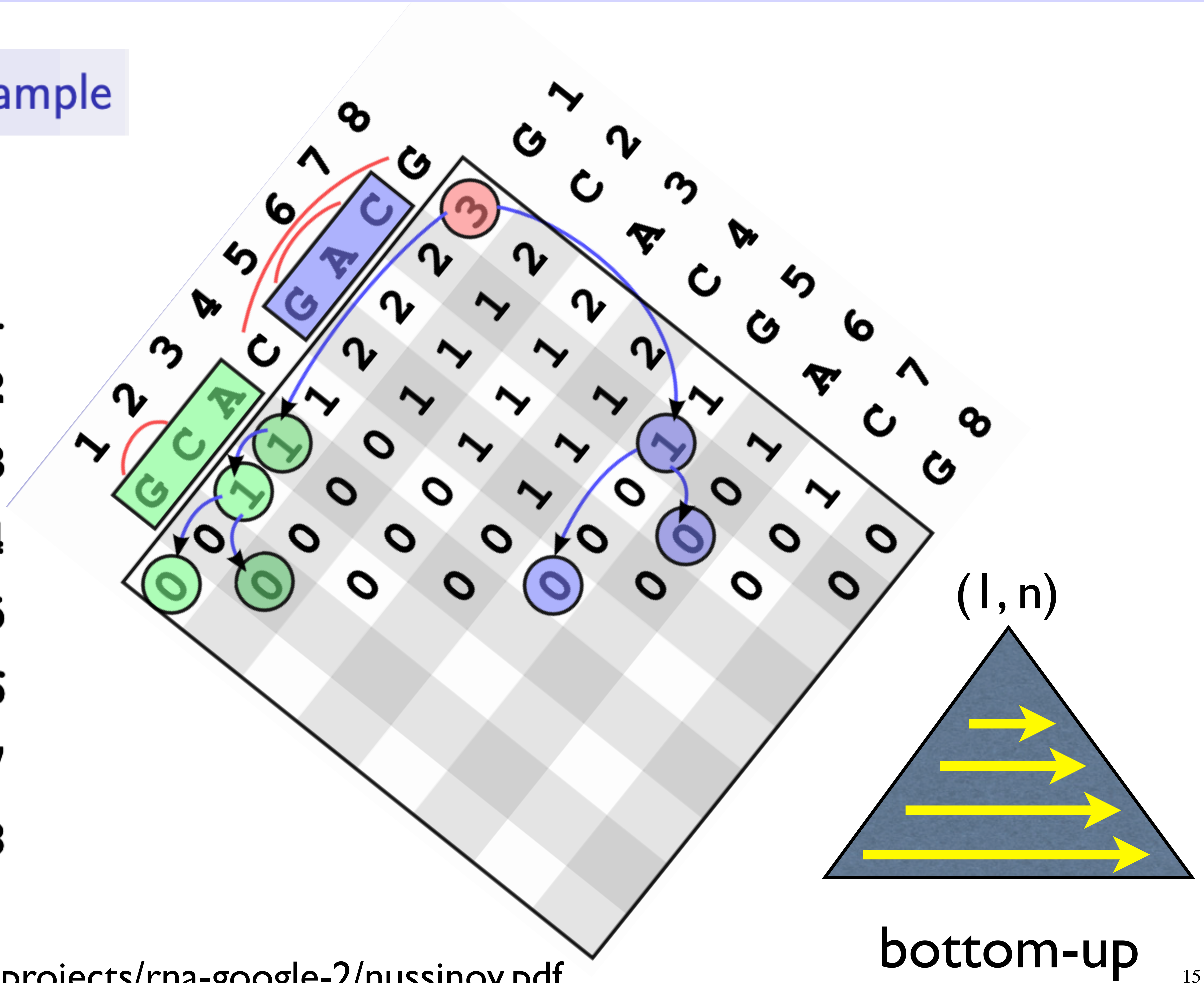


bottom-up left-to-right right-to-left

all $O(n^3)$

RNA Folding Example

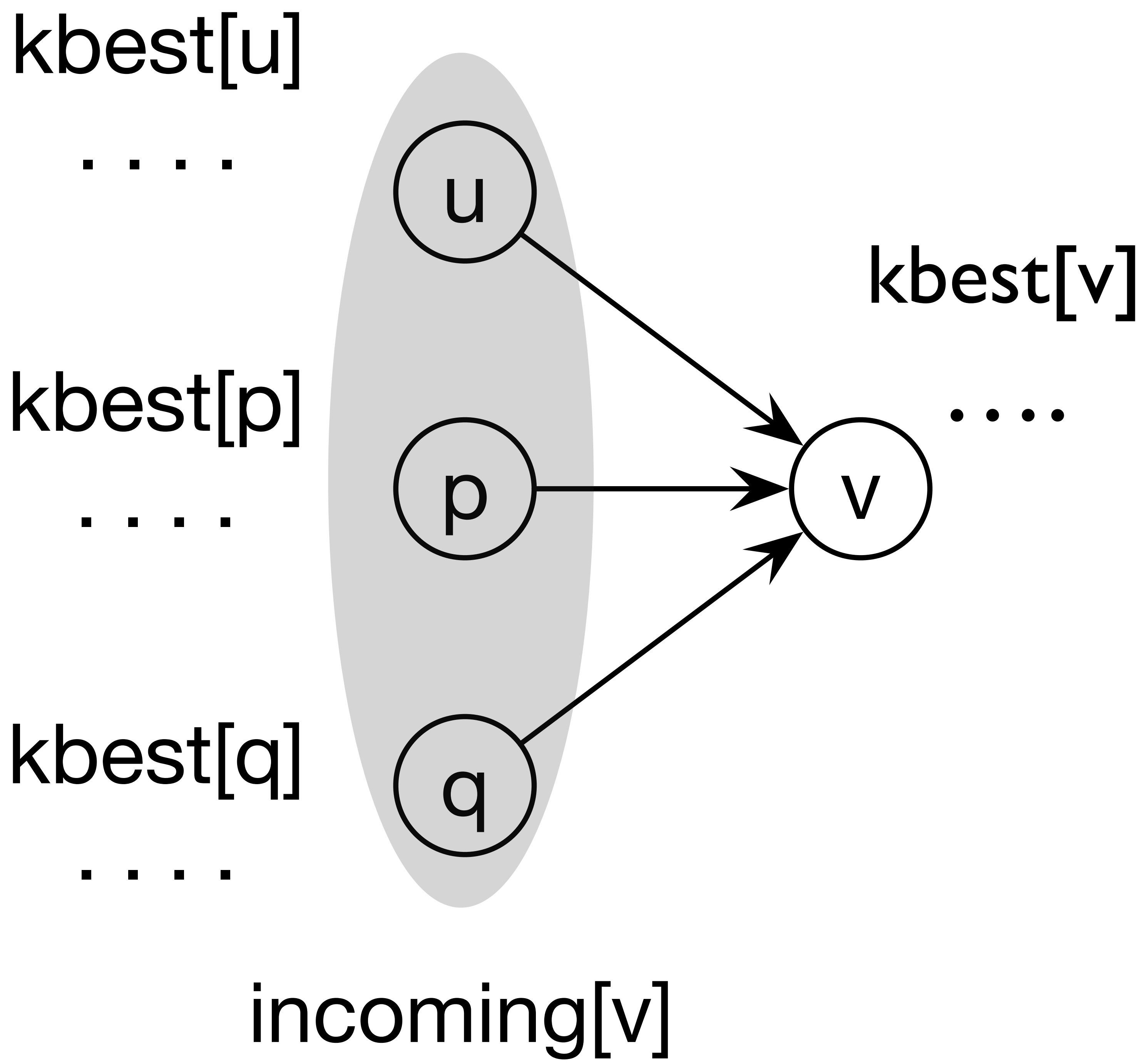




https://ad-publications.cs.uni-freiburg.de/student-projects/rna-google-2/nussinov.pdf

k-best Viterbi on Graph

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



cf. teams problem in HW4

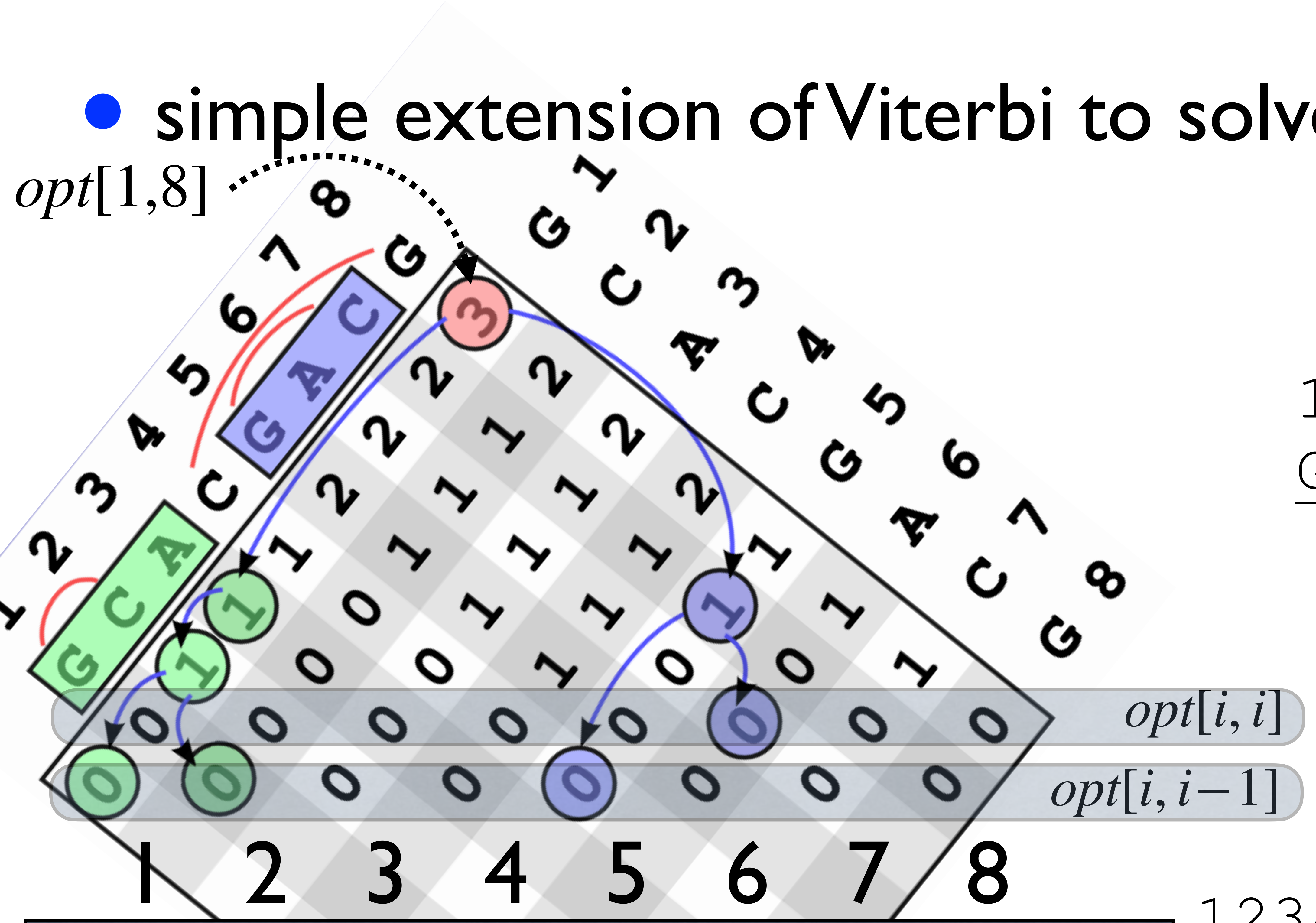
for each node v,
compute its kbest distances
from the kbest of each incoming node u

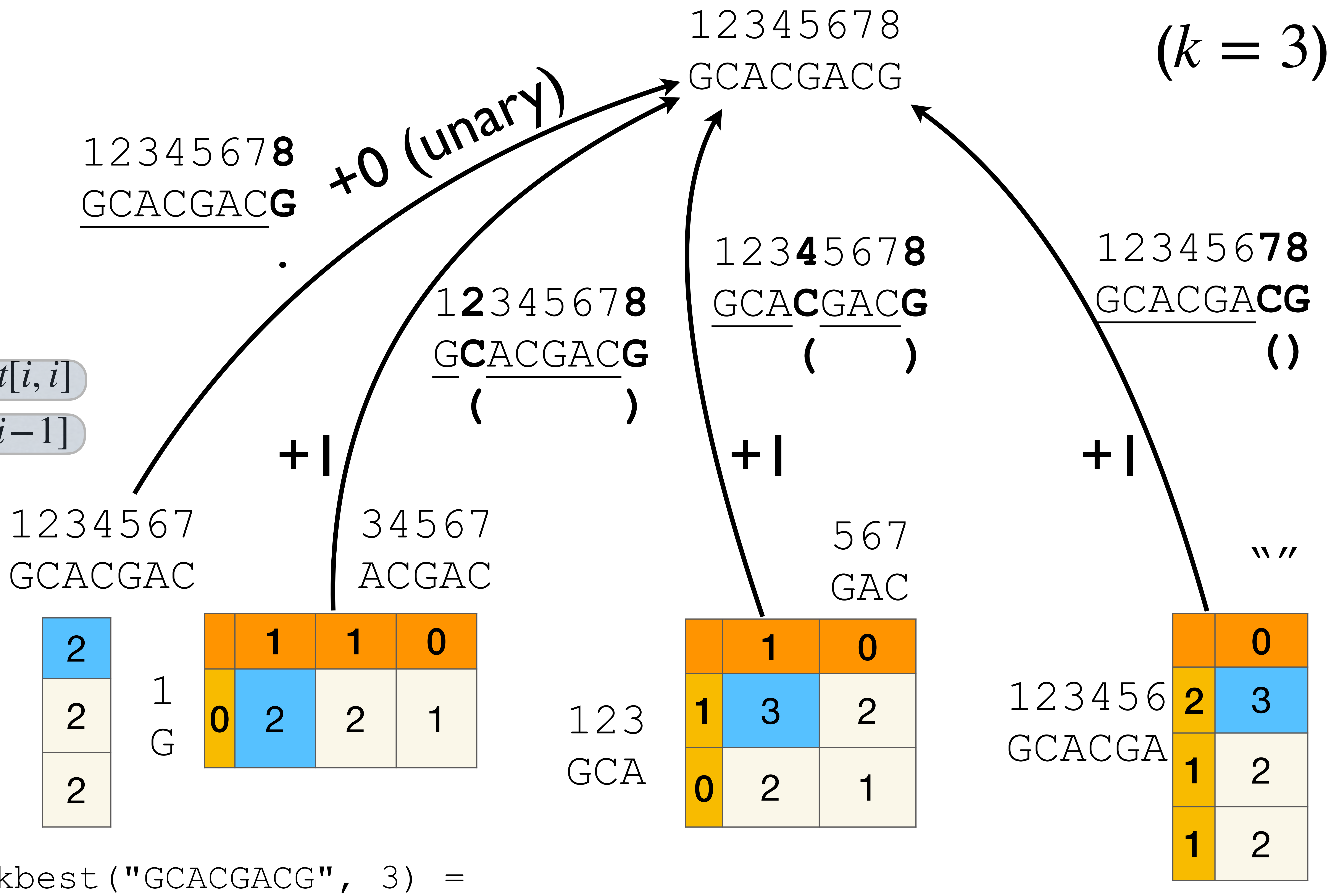
1-best: $O(E + V)$
k-best: $O(E + Vk \log d_{\max})$ where $d_{\max}$ is the max in-degree

can improve it to: (cf. midterm & teams, w/ quickselect)
k-best: $O(E + Vk \log k)$ (assume $k \ll d_{\max}$)
("most states do not have anybody on team USA")

k-best Viterbi on Hypergraph

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





$$opt[i,j] = \bigoplus \begin{cases} opt[i,j-1], \\ \bigoplus_{i \le p < j} (opt[i,p-1] \otimes opt[p+1,j-1] \otimes 1) \end{cases}$$

$$opt[i,i] = opt[i,i-1] = 1_{\otimes}$$

opt	⊕	⊗	$1_{\otimes}$
best	max	+	0
total	+	x	1

```
kbest("GCACGACG", 3) =
[(3, '().((.))'), (3, '().().()'), (2, '().()...')]
```