EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

g	6	6	t

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Dynamic Programming

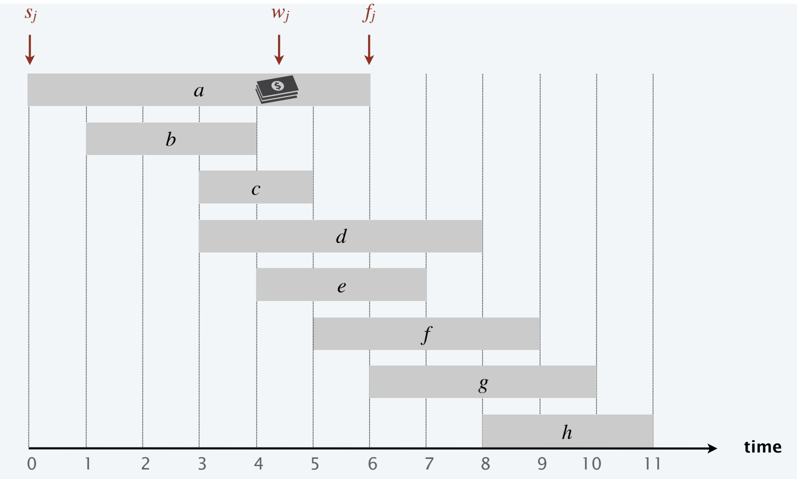
- Pioneered by Richard Bellman (late 1950s)
  - Extended significantly and is still seeing tremendous development
  - Most recent popular press applications: AlphaZero, AlphaGo, AlphaStar
- A general approach for breaking solutions of large problems into sequence of solutions of smaller problems
  - Originally developed for making decisions over time by decomposing the problem into making such decisions sequentially time by time
  - Dynamic programming: “planning over time”
- We have used dynamic programming already
  - Bellman-Ford, Floyd-Warshall
  - Want to study other applications of dynamic programming to understand technique

Applications of Dynamic Programming

- Partial list of other applications beyond networks
  - Seam carving in images
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models, for maximum-likelihood decoding
  - Knuth–Plass for word wrapping text in TeX
  - Parsing context-free grammars
  - Needleman–Wunsch/Smith–Waterman for sequence alignment
  - Railroad, UPS and Amazon delivery scheduling
  - Multi-move Games with perfect information (Chess, Checkers, Go, …)
  - Multi-move games with incomplete information (Poker, Stratego, …)
  - …

Weighted Interval Scheduling

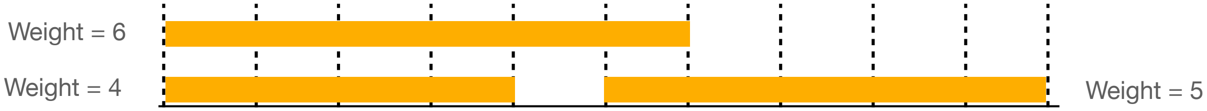
- A scheduling problem without greedy optimal solution
- Problem description: Given collection of intervals $I_1, \ldots, I_n$ with weights $w_1, \ldots, w_n$, choose a maximum weight set of non-overlapping intervals
  - Single machine scheduling of jobs with known start times, end times and values



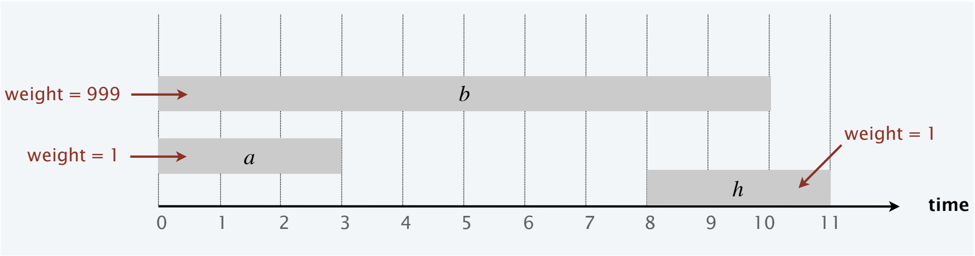
\

Weighted Interval Scheduling

- Greedy algorithm: Add jobs by weight, as long as they are compatible with previous schedule
    - Previous algorithm when duration was 1, had deadlines



- Greedy algorithm 2: Add jobs by earliest finish time
    - Previous algorithm when weights were all 1



Dynamic Programming Algorithm

- Assume intervals are indexed in increasing order of finishing time
  - Definition: p(j) is largest index of interval < j that can be scheduled with interval j
  - Finishing time of p(j) is before start time of j

- Smaller problem: what is the maximum value that can scheduled using only intervals from 1 to j? Define as Opt(j)
  - If j = 1, Opt(j) = w(1)
  - Recursion: use solution for j-1 to solve for j.
    - Either you schedule interval j in the optimal schedule, in which case you can only schedule intervals that finish before j starts,
    - Or you don't schedule interval j
  - Opt(j) = max(Opt(j-1), w(j) + Opt(p(j))

\

Dynamic Programming Algorithm

- Algorithm
  - Sort intervals 1, …, n in order of finishing time
  - For each interval j, compute p(j) by binary search
  - Recursively compute Opt(j) = max(Opt(j-1), w(j) + Opt(p(j)), for j = 1 to n, with initial condition Opt(0) = 0

- This can be really slow if you don’t use memory (e.g. solve recursively) because you keep computing Opt() for smaller values!

- Solution: store the previous values! Compute Opt(j) and store as a vector (memoization)

- Complexity: O(n log(n)): Sort n, plus n binary searches, plus n steps of O(1) updates

\

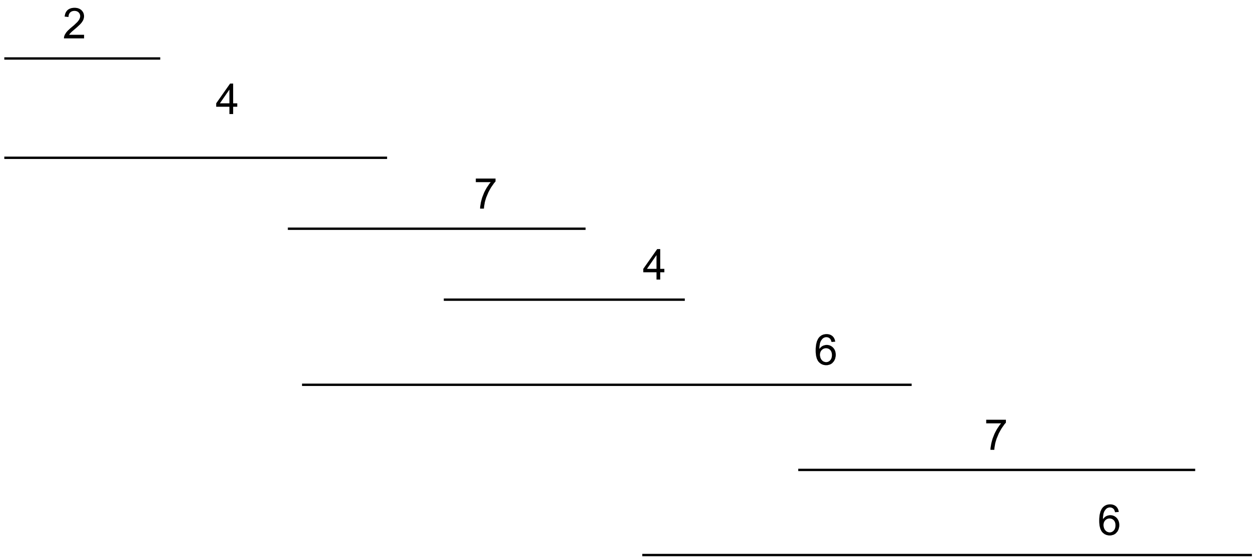
Dynamic Programming Algorithm

- This gives you the optimal value Opt(n)
  - How do we know which intervals were scheduled in optimal solution?
  - Similar problem to finding optimal path in shortest path problem: need extra information

- Can solve using a backward search
  - max = n, intervals = { }
  - While max > 0
    - If Opt(max) > Opt(max-1): this means interval in the optimal schedule
      - intervals $=$ intervals ∪ $\{I_n\}$, max = $p($max$)$
    - else: max = max - 1;

Example

j	p(j)	Opt(j)
1	0	2
2	0	4
3	1	9
4	2	9
5	1	9
6	4	16
7	3	16



- Intervals = {6, 3, 1}

\

Maximum Subarray Sum

- Given array A[1:n], find contiguous subarray A[j:k] with largest sum

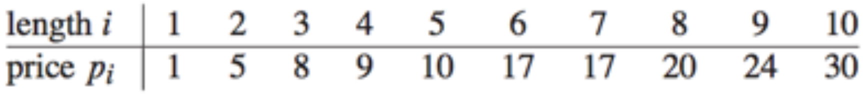
- Dynamic Programming:
  - MSE(k): maximum sum of a subarray ending at position k
  - MSE(1) = A[1]
  - MSE(k) = max(A[k], MSE(k-1) + A[k])

  -

i	1	2	3	4	5	6	7	8	9
A	-2	1	-3	4	-1	2	1	-5	4
MSE	-2	1	-2	4	3	5	6	1	5

Rod-Cutting

- A company buys long steel rods (of length n), and cuts them into shorter ones to sell
    - integral length only
    - cutting is free
    - rods of different lengths k sell for different price p_k



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Given n, what lengths should the rod be cut to maximize revenue?
    - n= 4, no cut has profit 9; cut into 2 and 2 has profit 10.
    - Brute force: list all integer partitions of n (there are many for large n…)
    - Better approach: Dynamic Programming

Rod-Cutting: Dynamic Programming

- Simple problem: solve for n = 1
- Define: Opt(j) = max profit for prod of length j
- Boundary value: Opt(0) = 0
- Recursion:

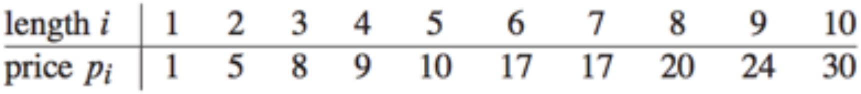
$$Opt(j+1) = max\left\{p_{j+1}, p_1 + Opt(j), p_2 + Opt(j-1), \cdots, p_j + Opt(1)\right\}$$

- Complexity: O(j) operations for step j

$$\sum_{j=1}^{n} j \in \Theta(n^2)$$

Example

\



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Opt(1) = 1; $Opt(2) = \max\{p_2, p_1 + Opt(1)\} = 5$
- $Opt(3) = \max\{p_3, p_2 + Opt(1), p_1 + Opt(2)\} = 8$
- $Opt(4) = \max\{9,9,10,9\} = 10$; $Opt(5) = \max\{10,11,13,13,11\} = 13$
- $Opt(6) = \max\{17,14,15,16,14,11\} = 17$
- $Opt(7) = \max\{17,18,18,18,17,15,18\} = 18$
- $Opt(8) = \max\{20,18,22,21,19,18,22,18\} = 22$
- $Opt(9) = \max\{24,23,22,25,22,20,25,22,23\} = 25$
- $Opt(10) = \max\{30,26,27,26,26,23,27,25,25,25\} = 30$

Knapsack Problem

\

- Given n items
  - Item j has value V(j) > 0, size c(j) > 0 (assume integer c(j))
- Given a box of size C > 0 (integer-valued)
- Select items that fit together in the box, and maximize the total value

$$\max_{x_i \in \{0,1\}} \sum_{i=1}^{n} V(i)\ x_i$$

Subject to the constraint $\sum_{i=1}^{n} c(i)\ x_i \le C$

x_i are indicator variables: 1 means item goes in the box, 0 item stays out

Select items that fit together in the box, and maximize the total value

\

Fractional Knapsack Problem

- Given n items
  - Item j has value V(j) > 0, size c(j) > 0
- Given a box of size C
- Select fractions items that fit together in the box, and maximize the total value

$$\max_{x_i \in [0,1]} \sum_{i=1}^{n} V(i)\ x_i$$

Subject to the constraint $\sum_{i=1}^{n} c(i)\ x_i \leq C$

x_i are fraction of item i that goes in the box

\

Fractional Knapsack Problem

- Greedy solution

  Rank items by diminishing value per unit size: $\frac{V(j)}{c(j)}$

  - Insert items in order; assume j is the last full item that fits in the box

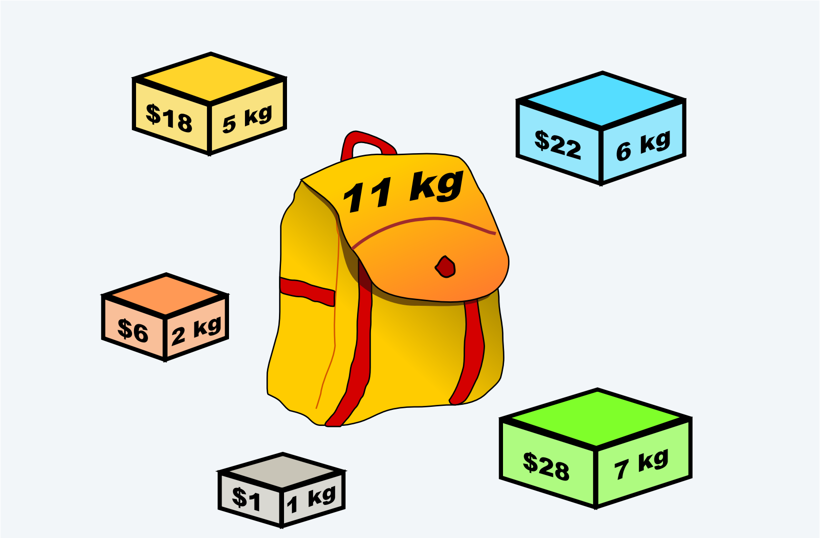
$$x_1, x_2, \ldots, x_j = 1;\;\; x_{j+1} = \left(C - \sum_{i=1}^{j} c(j)\right)/c(j+1)$$

  - Easy proof by contradiction; any solution that does not satisfy this can be replaced by a solution that satisfies this with at least as much value
  - Requires “partial credit”

Integer Knapsack Problem

- No partial credit for item scheduled partially
- Greedy algorithm no longer optimal

Item	V(j)	w(j)	V(j)/w(j)
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.67
5	28	7	4



- C = 11: Greedy {7,2,1} for value 35; {5,6} has value 40

Integer Knapsack : Dynamic Programming

- Value function: Opt(j,k) is best value considering items 1, …, j only, for capacity C = k

Easy initialization: $Opt(1,k) = \begin{cases} 0, & k < c(1); \\ V(1), & k \geq c(1) \end{cases}$

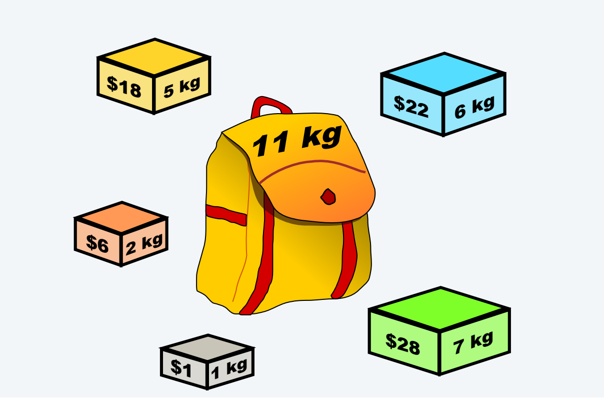
- If we consider an additional item j+1, for a capacity C = k, if we fit that item, then other items have to fit in remaining capacity C = k - c(j+1)
  - Recursion

$$Opt(j+1,k) = \begin{cases} Opt(j,k), & k < c(j+1) \\ \max\{Opt(j,k), V(k) + Opt(j, k - c(j+1))\} & \text{otherwise} \end{cases}$$

Example

\

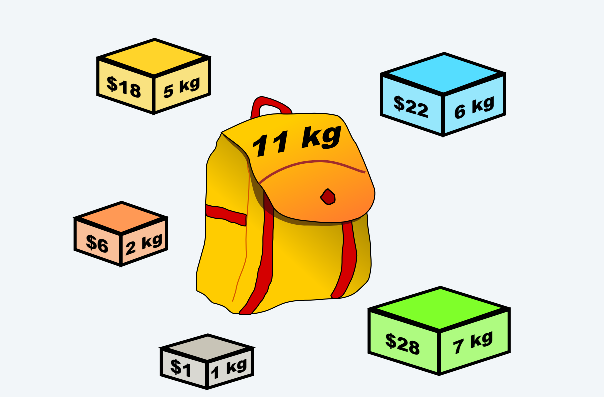
Item	V(j)	w(j)	V(j)/w(j)
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.67
5	28	7	4



j\k	0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	7	22	23	28	29	29	40
5	0	1	6	7	7	7	22	28	29	34	35	40

Example: What is in the Bag?

Item	V(j)	w(j)	V(j)/w(j)
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.67
5	28	7	4



j\k	0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	7	22	23	28	29	29	40
5	0	1	6	7	7	7	22	28	29	34	35	35

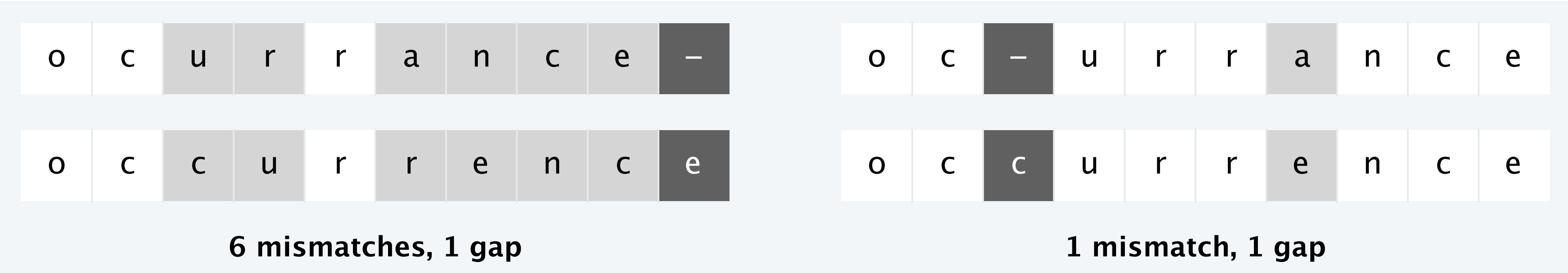
Integer Knapsack Complexity

\

- Complexity: $\Theta(nC)$
  - Number of rows: n
  - Number of columns: C
  - Computation per entry: O(1)
- Complexity is not polynomial (depends on C, so pseudo-polynomial)
- Space required is also $\Theta(nC)$
- Note algorithm depends critically on the fact that sizes c(j) are integers
  - Can handle non-integer c(j) with a lot more notation

Sequence Alignment

- How similar are two sequences of symbols?
  - Example: ocurrance and occurrence



o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	–	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

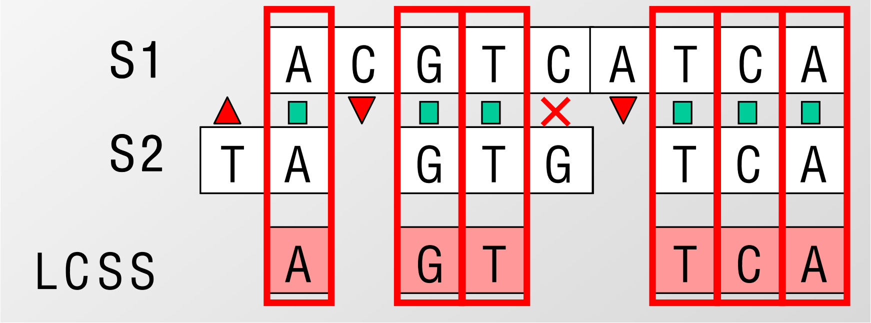
1 mismatch, 1 gap

- Applications: Bioinformatics, spell correction, machine translation, speech recognition, information extraction

Comparing Two DNA sequences

- Given two strings, what is the best match?

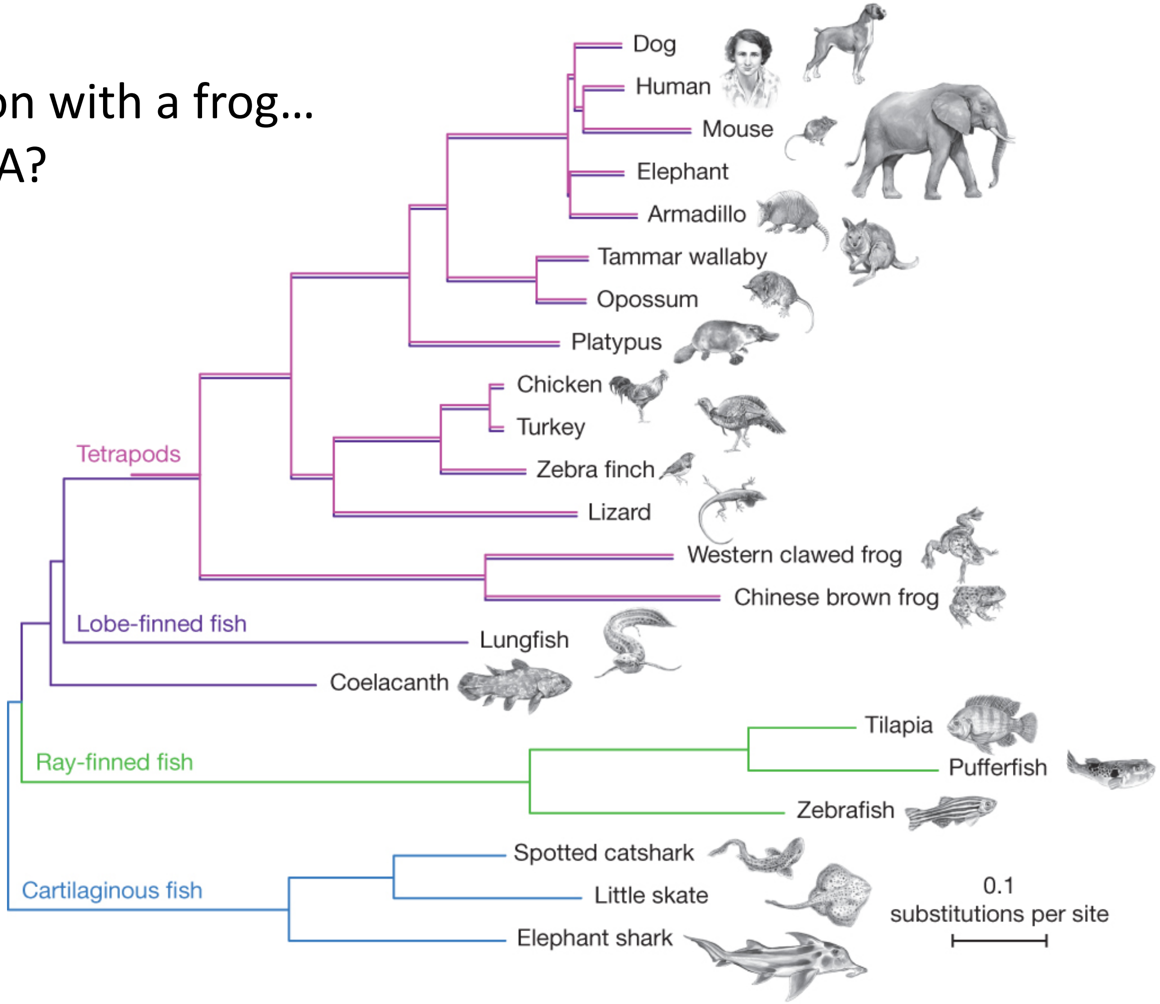
S1	A	C	G	T	C	A	T	C	A
S2	T	A	G	T	G	T	C	A	



S1		A	C	G	T	C	A	T	C	A
	▲	■	▼	■	■	✕	▼	■	■	■
S2	T	A		G	T	G		T	C	A
LCSS		A		G	T			T	C	A

Why Do We Care?

- You have lots in common with a frog… which parts of your DNA?



Example Problem

- Given two strings $x = x_1x_2\cdots x_m$, $y = y_1y_2\cdots y_n$

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

- an alignment is an assignment of gaps to positions 0,..., m in x, and 0,..., n in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence and there are no crossings
  - No crossings —> if j matched with k, and j' > j matched with k', then k' > k

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

What is a Good Alignment?

```
AGGCTAGTT
AGCGAAGTTT
```

- Alignment 1: 6 matches, 3 mismatches, 1 gap

```
AGGCTAGTT
AGCGAAGTTT
```

- Alignment 2: 7 matches, 1 mismatch, 3 gaps

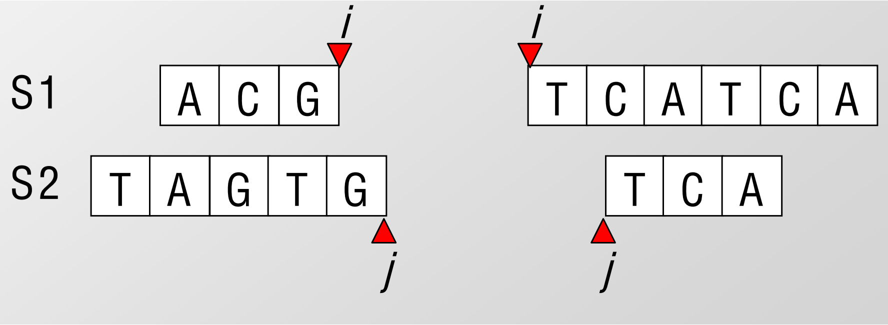
```
AGGCTA-GTT-
AG-CGAAGTTT
```

- Alignment 3: 7 matches, 0 mismatches, 5 gaps

```
AGGC-TA-GTT-
AG-CG-AAGTTT
```

Edit Distance

- Concept due to Levenshtein 1966, Needleman–Wunsch 1970
- Scoring function
  - Cost of mutation (mismatch)
    - s(x,y) is cost of matching $x \neq y$
  - Cost of insertion/deletion
    - δ is cost of matching x to a gap, or matching y to a gap
  - Reward of correct match
    - s(x, y) is value of correctly matching when x = y



- Complex search problem
  - For sequences of length 100, number of possible matches is $9 \cdot 10^{58}$

Match and Mismatch Rewards

- BLOcks SUbstitution Matrix (BLOSUM): A 20x20 table amino-acid scoring table based on observation of protein mutation rates
  - Gives the score of aligning amino-acid X with amino-acid Y (-s(x,y)

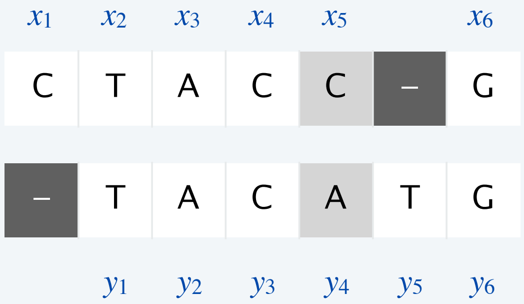
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4
	Ala	**Arg**	**Asn**	**Asp**	**Cys**	**Gln**	**Glu**	**Gly**	**His**	**Ile**	**Leu**	**Lys**	**Met**	**Phe**	**Pro**	**Ser**	**Thr**	**Trp**	**Tyr**	**Val**

Cost of Alignment

- Cost of M is

$$Cost(M) = \sum_{i,j \text{ matched}} s(x_i, y_j) + \sum_{i:x_i \text{ unmatched}} \delta + \sum_{j:y_j \text{ unmatched}} \delta$$

- Useful structure: score is additive —> similar structure to path length as additive over edge lengths
  - For a given split (i,j), we have score of best alignment x[1:n] and y[1:n] is sum of scores of best alignment x[1:i], y[1:j] + best alignment x[i+1:m], y[j+1:n]
  - This will allow us to use dynamic programming


x_1 x_2 x_3 x_4 x_5 x_6
C T A C C – G
– T A C A T G
y_1 y_2 y_3 y_4 y_5 y_6

an alignment of CTACCG and TACATG

$M = \{ x_2–y_1, x_3–y_2, x_4–y_3, x_5–y_4, x_6–y_6 \}$

```
Spokesperson confirms     senior government adviser was found
Spokesperson said     the senior            adviser was found
```

Dynamic Programming (Needleman-Wunsch) \

- Let OPT(i, j) = minimum cost of aligning prefix strings $x_1 x_2 \cdots x_i,\ y_1 y_2 \cdots y_j$
- Goal. Is to compute OPT(m,n)
- Idea: Assume we know OPT(i-1, j-1), OPT(i,j-1), and OPT(i-1,j-1):
  - Case 1. OPT(i, j) matches $x_i \to y_j$: $Opt(i, j) = s(x_i, y_j) + OPT(i-1, j-1)$
  - Case 2a. OPT(i, j) leaves x_i unmatched: $Opt(i, j) = \delta + OPT(i-1, j)$
  - Case 2b. OPT(i, j) leaves y_j unmatched: $Opt(i, j) = \delta + OPT(i, j-1)$
- Initially, $Opt(i,0) = i\delta; \quad Opt(0,j) = j\delta$
- Iteration:

$$Opt(i, j) = \min \left\{ s(x_i, y_j) + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1) \right\}$$

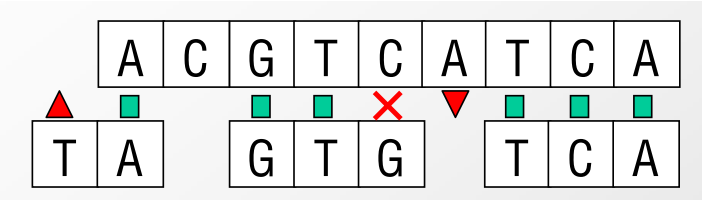
Ptr(i,j) = {diag, up, left} corresponding to which term is minimized

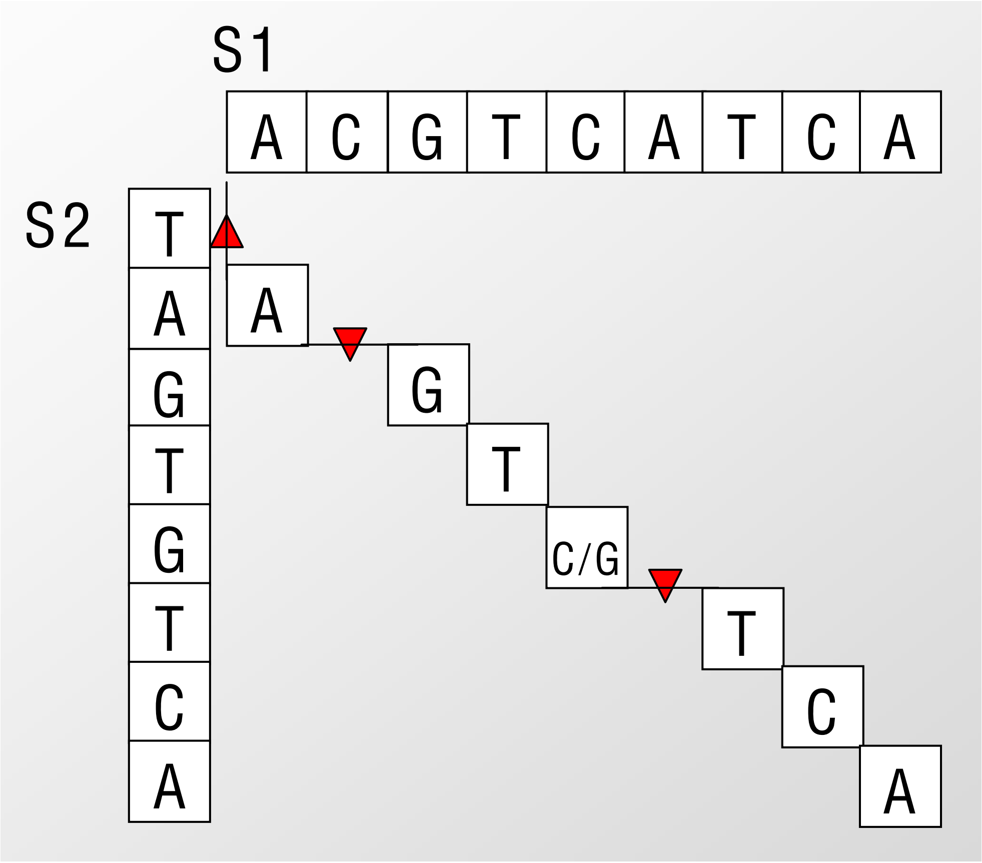
Proof of Correctness

\

- Create a grid graph with vertices (i,j), i= 0, …, m, j = 0, …, n
  - Edges from vertices (i-1,j) to (i,j) with weight δ
  - Edges from vertices (i,j-1) to (i,j) with weight δ
  - Edges from vertices (i-1,j-1) to (i,j) with weight $s(x_i, y_j)$
- Note: Graph is acyclic
- The problem is to find a shortest path from (0,0) to (m,n) in this graph!
  - OPT(i,j) is shortest distance from (0,0) to (i,j)
  - Needleman-Wunsch is Bellman-Ford!
- Bellman-Ford finds shortest path in acyclic graphs with negative weights

Matrix Representation of Alignment





Small Example

OPT(i,j) with $\delta = 2;\ s(x_i, y_j) = \begin{cases} 2, & x_i \neq y_j \\ -1, & x_i = y_j \end{cases}$

OPT =

	-	A	G	C
-	0	2	4	6
A	2	-1	1	3
A	4	1	1	3
A	6	3	3	3
C	8	5	5	2

PTR =

	-	A	G	C
-	0	Left	Left	Left
A	Up	Diag	Left	Left
A	Up	Diag	Diag	Diag
A	Up	Diag	Diag	Diag
C	Up	Up	Up	Diag

Mismatch = -1
Match = 2

Example

	j	0	1	2	3	4	5
i			c	a	d	b	d
0		0	-1	-2	-3	-4	-5
1	a	-1					
2	c	-2					
3	b	-3					
4	c	-4					
5	d	-5					
6	b	-6					

←T

↑
S

c
-

Score(c,-) = -1

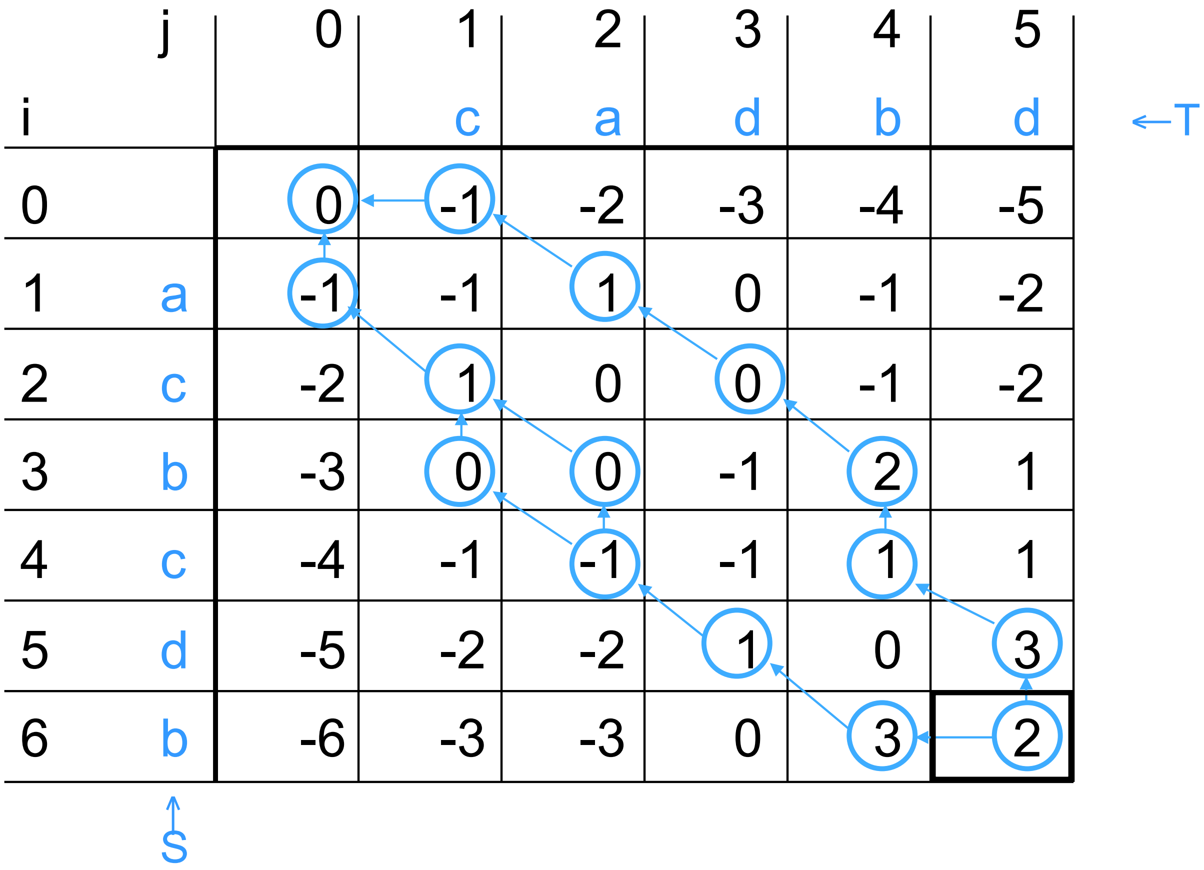
Mismatch = -1
Match = 2

Example

	j	0	1	2	3	4	5	
i			c	a	d	b	d	←T
0		0	-1	-2	-3	-4	-5	
1	a	-1	-1	1	0	-1	-2	
2	c	-2	1	0	0	-1	-2	
3	b	-3	0	0	-1	2	1	
4	c	-4	-1	-1	-1	1	1	
5	d	-5	-2	-2	1	0	3	
6	b	-6	-3	-3	0	3	2	
	↑ S							

Optimal Match: Backtrack Pointers

\

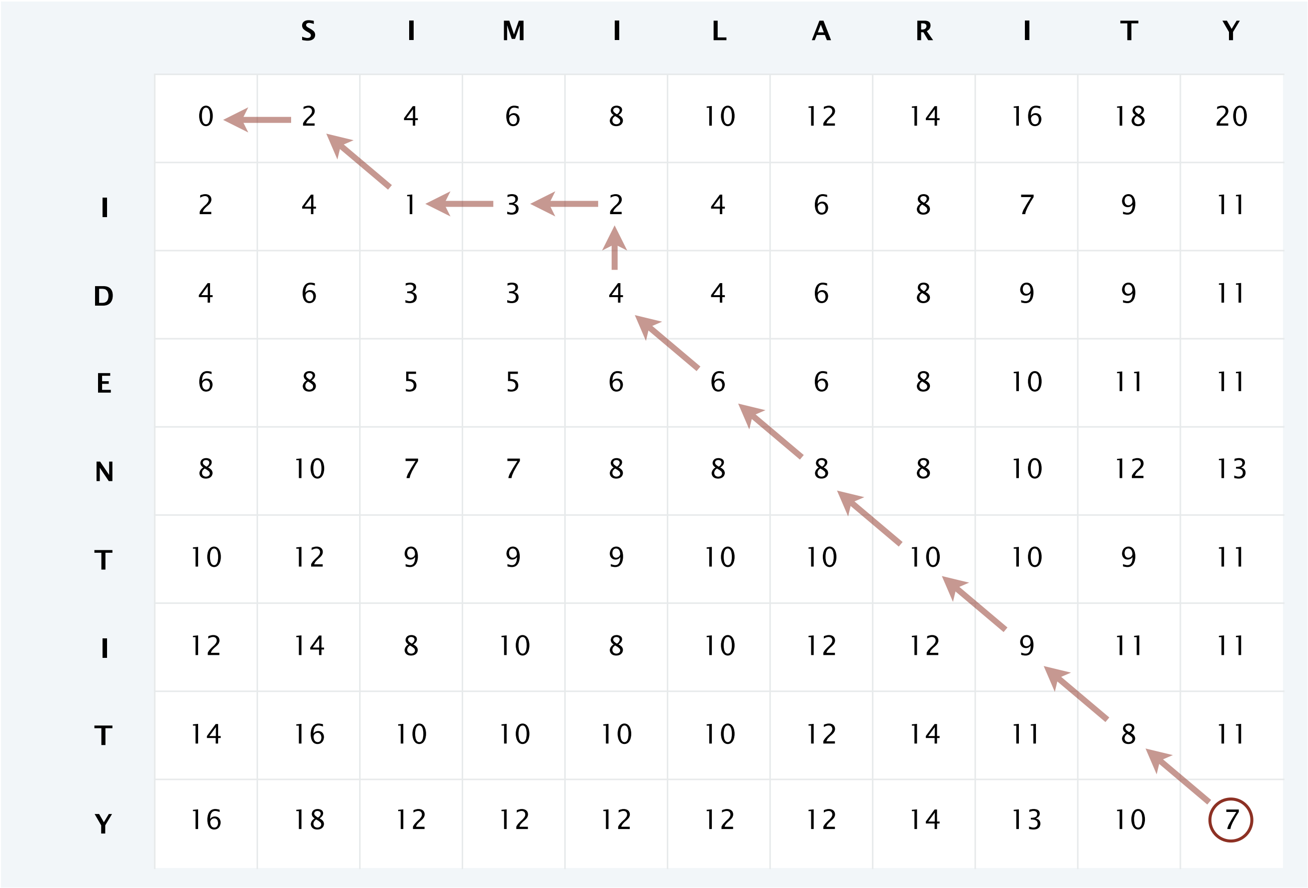


j		0	1	2	3	4	5
i			c	a	d	b	d
0		0	-1	-2	-3	-4	-5
1	a	-1	-1	1	0	-1	-2
2	c	-2	1	0	0	-1	-2
3	b	-3	0	0	-1	2	1
4	c	-4	-1	-1	-1	1	1
5	d	-5	-2	-2	1	0	3
6	b	-6	-3	-3	0	3	2

A Larger Example

$$\delta = 2;$$

$$s(x_i, y_j) = \begin{cases} 2, & x_i \neq y_j \\ -1, & x_i = y_j \end{cases}$$

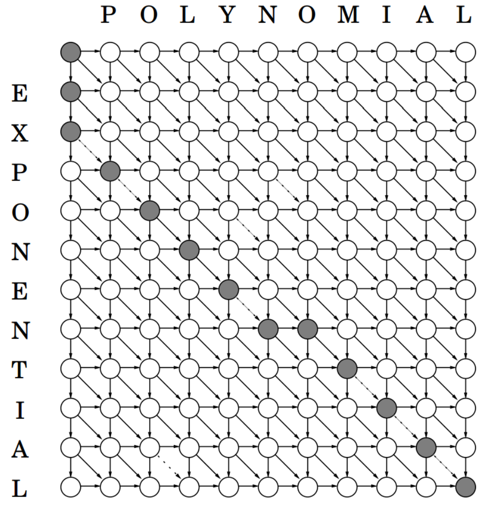


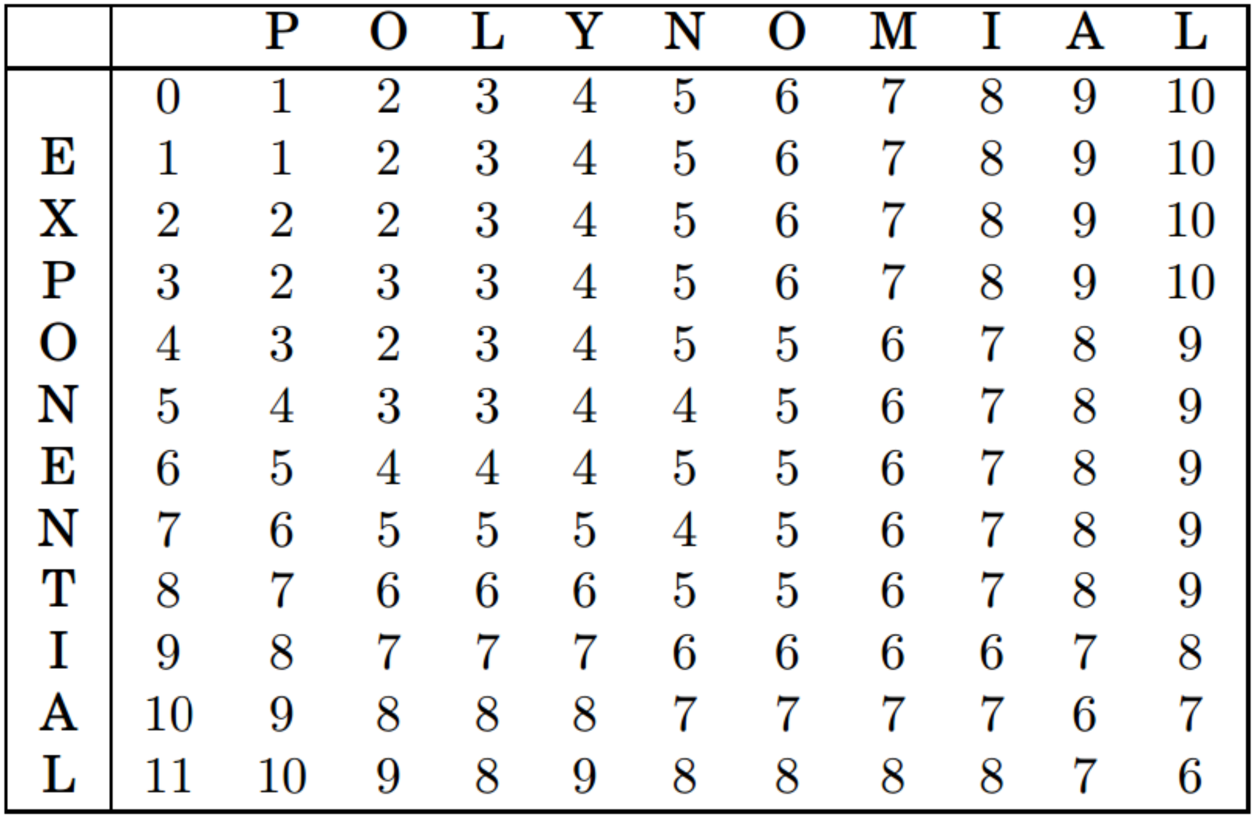
		S	I	M	I	L	A	R	I	T	Y
	0	2	4	6	8	10	12	14	16	18	20
I	2	4	1	3	2	4	6	8	7	9	11
D	4	6	3	3	4	4	6	8	9	9	11
E	6	8	5	5	6	6	6	8	10	11	11
N	8	10	7	7	8	8	8	8	10	12	13
T	10	12	9	9	9	10	10	10	10	9	11
I	12	14	8	10	8	10	12	12	9	11	11
T	14	16	10	10	10	10	12	14	11	8	11
Y	16	18	12	12	12	12	12	14	13	10	7

Another Example

$\delta = 1;$

$$s(x_i, y_j) = \begin{cases} 1, & x_i \neq y_j \\ 0, & x_i = y_j \end{cases}$$





		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Sequence Matching Complexity

- Need to complete table of m by n
  - Length of x: m, length of y: n
- Computational complexity O(mn)
  - O(1) operations to compute new element
  - Polynomial!
- Still, may be too slow for long DNA sequences
  - 50,000 genes…
- Search for faster approximate algorithms