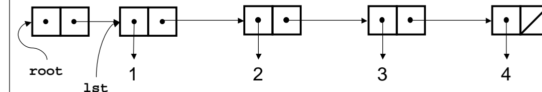
Important Example: Gene Sequence Matching

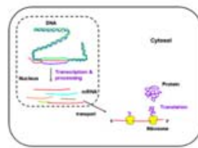
- "Century of Biology"
- Two views of computer science's relationship to biology:
  – Bioinformatics: computational methods to help discover new biology from lots of data
  – Engineering on biological substrates: using biology instead of electronics

Corrigiendum

```scheme
(define (filter! f lst)
  (if (null? lst)
      '()
      (let ((root (cons '() lst)))
        (define (iter l)
          (cond ((null? (cdr l)))
                ((f (cadr l)) (iter (cdr l)))
                (else
                 (set-cdr! l (cddr l))
                 (iter l))))
        (iter root)
        (cdr root))))
```



Central Dogma of Modern Biology


Kuo, JBI 37 (2004) 293–303

- DNA encodes genes & is inherited
- DNA is *transcribed* under control of proteins into RNA
- RNA is *translated* into proteins by ribosomes
- Proteins run the cell, and thus organisms

Recursion!

Genetics

- Proteins are assembled from a sequence of amino acids
- DNA (and RNA) represent each amino acid by a triple of codes in a 4-letter "alphabet" of nucleotides: guanine, cytosine, adenine and thymine (uracil in RNA), which we denote by G, C, A and T
- How many amino acids could we encode?

$4^3 = 64$

How Nucleotides code for Amino Acids

SECOND POSITION

FIRST POSITION	U	C	A	G	THIRD POSITION
U	phenyl-alanine	serine	tyrosine	cysteine	U
					C
	leucine		stop	stop	A
			stop	tryptophan	G
C	leucine	proline	histidine	arginine	U
					C
			glutamine		A
					G
A	isoleucine	threonine	asparagine	serine	U
					C
			lysine	arginine	A
	* methionine				G
G	valine	alanine	aspartic acid	glycine	U
					C
			glutamic acid		A
					G

* and start

From http://web.mit.edu/esgbio/www/dogma/dogma.html

Biological Matching Heuristic

- Evolution makes incremental changes in genome
  – Point mutations: e.g., C ==> A
  – Dropping or inserting a nucleotide during copying
  – Gene duplication, then
    - Separate drift
    - Separate function
- Similarity of sequence useful to discover
  – Similarity of function
  – Evolutionary history
- Given a gene sequence with unknown function, match it against ones with known function to get clues

Define a Distance Metric

- Given two sequences, s1 & s2,
  - Distance is 0 if they are identical
  - Penalty for each point mutation
    - Different for different mutations
  - Penalty for insertion/deletion of nucleotides
  - "Distance" is sum of penalties
    - May be asymmetric, so not true distance
- Corresponds to log(probability), assuming independence of each mutation

Matching Paradigm

- Given a new section of DNA of interest, match it against all potentially useful segments of known-function DNA
- Best match(es) may lead to insight

≈ Needleman-Wunch, Smith-Waterman Algorithms

Details of matching

```
a a t c t g c c t a t t g t c g a c g c
        m     m m m   m m               m
a a t c a g c a g c t c a t c g a c g g

a a t c a g c a g c t c a t c g a c g g
  m m m m m m m m
a g a t c a g c a c t c a t c g a c g g
```

or

```
a a t c a g c a g c t c a t c g a c g g
          ...     ^
a g a t c a g c a c t c a t c g a c g g
```


Representing Mutation Penalty

	A	C	G	T
A	0	.3	.4	.3
C	.4	0	.2	.3
G	.1	.3	0	.2
T	.3	.4	.1	0

2-D Table

```
(define point-mutations (make-table2))
(table2-set! point-mutations 'A 'A 0)
(table2-set! point-mutations 'A 'C .3)
(table2-set! point-mutations 'A 'G .4)
...

(table2-get point-mutations 'A 'C)
>> .3
(table2-get point-mutations 'A 'X)
>> #f
```

- But how to implement a 2-D table?

Remember Table Abstractions

```
(define (find-assoc-binding key alist)
  (cond ((null? alist) #f)
        ((equal? key (caar alist)) (car alist))
        (else (find-assoc-binding key (cdr alist)))))

(define (find-assoc key alist)
  (let ((binding (find-assoc-binding key alist)))
    (if binding
        (cadr binding)
        #f)))

(define (add-assoc key val alist)
  (cons (list key val) alist))
```

Note Scheme's
assoc
assv
assq

Non-Abstract but Compact!

```
(define mutation-penalties
  '((a        (c .3) (g .4) (t .3))
    (c (a .4)        (g .2) (t .3))
    (g (a .1) (c .3)        (t .2))
    (t (a .3) (c .4) (g .1)       )))

(define (mutation to from)
  (if (eq? from to)
      0
      (let ((row (find-assoc-binding ; == assoc
                   to mutation-penalties)))
        (if row
            (find-assoc from (cdr row)); == cadr of assoc
            #f))))
```


Remember Table ADT

```
(define table1-tag 'table1)
(define (make-table1)
  (cons table1-tag '()))
(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))
(define (table1-set! tbl key val)
  (set-cdr! tbl
            (add-assoc! key val (cdr tbl))))
```

- Note: we mutate structure, unlike before.

Mutating Version of add-assoc

```
(define (add-assoc! key val alist)
  (let ((binding (find-assoc-binding key alist)))
    (cond (binding
           (set-car! (cdr binding) val)
           alist)
          (else
           (add-assoc key val alist)))))
```


Table2 is a table of Table1's

```
(define table2-tag 'table2)
(define (make-table2)
  (cons table2-tag (make-table1)))

(define (table2-get tbl key-row key-col)
  (let ((row (table1-get (cdr tbl) key-row)))
    ;; row is itself a table1!
    (if row (table1-get row key-col) #f)))

(define (table2-set! tbl key-row key-col val)
  (let ((row (table1-get (cdr tbl) key-row)))
    (if row
        (table1-set! row key-col val)
        (let ((new-row (make-table1)))
          (table1-set! new-row key-col val)
          (table1-set! (cdr tbl) key-row new-row)))))
```


Defining Mutations More Abstractly

```
(table2-set! point-mutations 'a 'a 0)
(table2-set! point-mutations 'a 'c 0.3) ;; e.g., from c to a
(table2-set! point-mutations 'a 'g 0.4)
(table2-set! point-mutations 'a 't 0.3)
(table2-set! point-mutations 'c 'a 0.4)
(table2-set! point-mutations 'c 'c 0)
(table2-set! point-mutations 'c 'g 0.2)
(table2-set! point-mutations 'c 't 0.3)
(table2-set! point-mutations 'g 'a 0.1)
(table2-set! point-mutations 'g 'c 0.3)
(table2-set! point-mutations 'g 'g 0)
(table2-set! point-mutations 'g 't 0.2)
(table2-set! point-mutations 't 'a 0.3)
(table2-set! point-mutations 't 'c 0.4)
(table2-set! point-mutations 't 'g 0.1)
(table2-set! point-mutations 't 't 0)
```


We have the Penalties

```
point-mutations
>>
(table2
  table1
  (t (table1 (t 0) (g 0.1) (c 0.4) (a 0.3)))
  (g (table1 (t 0.2) (g 0) (c 0.3) (a 0.1)))
  (c (table1 (t 0.3) (g 0.2) (c 0) (a 0.4)))
  (a (table1 (t 0.3) (g 0.4) (c 0.3) (a 0))))

(define omit-penalty .5)
(define insert-penalty 0.7)
```

Simplest Matcher

... slooooooooooooow!!!

$\approx 3^n$

```
(define (match0 one two)
  (define (helper x y score)
    (cond ((and (null? x) (null? y)) score)
          ((null? x)
           (helper x (cdr y) (+ score omit-penalty)))
          ((null? y)
           (helper (cdr x) y (+ score insert-penalty)))
          ((eq? (car x) (car y))
           (helper (cdr x) (cdr y) score))
          (else
           (let ((mutated
                  (helper (cdr x) (cdr y)
                          (+ score (mutation (car x) (car y)))))
                 (omitted
                  (helper x (cdr y) (+ score omit-penalty)))
                 (inserted
                  (helper (cdr x) y (+ score insert-penalty))))
             (min mutated omitted inserted)))))
  (helper one two 0.0))
```


Remember Fibonacci

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

$T(n) = \Theta(\phi^n)$

```
(define old-vals (make-table1))

(define (fibmemo n)
  (let ((old-val (table1-get old-vals n)))
    (cond (old-val old-val)
          (else
           (let ((new-val
                  (cond ((= n 0) 0)
                        ((= n 1) 1)
                        (else (+ (fibmemo (- n 1))
                                 (fibmemo (- n 2)))))))
             (table1-set! old-vals n new-val)
             new-val)))))
```

$T(n) = \Theta(n)$

Better Memoized Matching

```
(define (match1 one two)
  (let ((past (make-table2)))
    (define (helper x y score)
      (let ((old (table2-get past x y)))
        (if old (+ old score)
            (let ((new
                   <<guts of match0's helper>>))
              (table2-set! past x y (- new score))
              new))))
    (helper one two 0.0)))
```

$T(n) = \Theta(n^2)$

- We store best score from *here* (x,y) to end.
- Still too slow for long sequences!
- Can we *not* consider some of the worst partial matches?

Cutting off Bad Paths

- Estimate an upper bound on matches of interest
- Declare any match worse than this to be infinitely bad (and stop pursuing it)

```
(cond
  ((> score limit) infty)
  ((and (null? x) (null? y)) score)
  ...
```

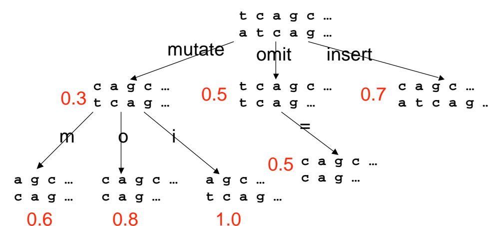

Performance with Cutoff

penalty (#past)

	infinity	15	10	5	3	2.5	2	1.5
s1, s2	1.8 (243)	1.8 (243)	1.8 (243)	1.8 (204)	1.8 (117)	1.8 (92)	1.8 (78)	XXX (50)
s1s1, s2s2	3.6 (1025)	3.6 (999)	3.6 (777)	3.6 (386)	XXX (173)	XXX (120)	XXX (78)	XXX (50)
s1s1s1s1, s2s2s2s2	7.2 (4242)	7.2 (2435)	7.2 (1496)	XXX (449)	XXX (173)	XXX (120)	XXX (78)	XXX (50)

- Dramatic savings
- May miss answer if we guess cutoff badly
- Better: just pursue the "top *k*" paths
  - Reformulate as a *search problem*

Idea: Pursue "best" matches



Generalized Search

- ***Depth-first***: pursue first alternative from last "move"
  - E.g., first consider string of all mutations
- ***Breadth-first***: consider all "moves" of length 1, then 2, then 3, …
  - Must keep track of all possible move sequences
- ***Best-first***: consider extending only the best sequence
  - Still must keep track of all
- ***Beam***: like best-first, but remember only top *n* sequences

Search State

```
(define (make-search-state
          score x y history)
  (list score x y history))
(define ss-score car)
(define ss-x cadr)
(define ss-y caddr)
(define ss-history cadddr)
```

```
(define (successors ss)
  (let ((x (ss-x ss))
        (y (ss-y ss))
        (s (ss-score ss))
        (h (ss-history ss)))
    (if (eq? (car x) (car y))
        (list (make-search-state
               s (cdr x) (cdr y)
               (cons (list '= (car x) (car y)) h)))
        (list (make-search-state
               (+ s (mutation (car x) (car y)))
               (cdr x) (cdr y)
               (cons (list 'm (car x) (car y)) h))
              (make-search-state
               (+ s omit-penalty) x (cdr y)
               (cons (list 'o (car y)) h))
              (make-search-state
               (+ s insert-penalty) (cdr x) y
               (cons (list 'i (car x)) h))))))
```

+ special cases when x or y are null

Framework for Search

```
(define (search start-state done? succ-fn merge-fn)
  (define (inner queue)
    (if (null? queue)
        #f
        (let ((current (car queue)))
          (if (done? current)
              current
              (inner
               (merge-fn (succ-fn current)
                         (cdr queue)))))))
  (inner (list start-state)))
```

- Have we reached "goal"?
- Order in which to explore moves
- What "moves" can we make from current state?

Depth-first

```
(define (completed? ss)
  (and (null? (ss-x ss)) (null? (ss-y ss))))

(define (match-dfs x y)
  (search (make-search-state 0.0 x y '())
          completed?
          successors
          append))

>(match-dfs '(a a t c t g c c t a t t g t c g a c g c)
            '(a a t c a g c a g c t c a t c g a c g g))
(1.8 () () ((m c g) (= g g) (= c c) (= a a)
  (= g g) (= c c) (= t t) (m g a) (m t c)
  (= t t) (m a c) (m t g) (m c a) (= c c)
  (= g g) (m t a) (= c c) (= t t) (= a a)
  (= a a)))
```

- Will find left-most, not necessarily best!

Breadth-first

```
(define (match-dfs x y)
  (search (make-search-state
           0.0 x y '())
          completed?
          successors
          append))
```

```
(define (match-bfs x y)
  (search (make-search-state
           0.0 x y '())
          completed?
          successors
          (lambda (new old)
            (append old new))))
```

- Finds best answer, but with maximum work
- For *n* mismatches, BFS will generate 3^n states on the queue
- Won't find *any* solution until all partial matches have been put on queue

Best-first

```
(define (better? ss1 ss2)
  (< (ss-score ss1) (ss-score ss2)))

(define (priority-merge new old)
  (let ((new-sorted (sort new better?)))
    (define (merge x y ans)
      (cond ((and (null? x) (null? y))
             ans)
            ((null? x)
             (append (reverse! y) ans))
            ((null? y)
             (append (reverse! x) ans))
            ((better? (car x) (car y))
             (merge (cdr x) y (cons (car x) ans)))
            (else
             (merge x (cdr y) (cons (car y) ans)))))
    (reverse! (merge new-sorted old '()))))
```

Already sorted!

Best-first examples

```
(define (match-best-fs x y)
  (search (make-search-state
           0.0 x y '())
          completed?
          successors
          priority-merge))
```

```
(define s1 '(a a t c t g c c t a t t g t c g a c g c))
(define s2 '(a a t c a g c a g c t c a t c g a c g g))
(define s3 '(a g a t c a g c a c t c a t c g a c g g))
```

```
> (match-best-fs s1 s2)
(1.8 () ()
 ((m c g) (= g g) (= c c) (= a a)
 (= g g) (= c c) (= t t) (m g a)
 (m t c) (= t t) (m a c) (m t g)
 (m c a) (= c c) (= g g) (m t a)
 (= c c) (= t t) (= a a) (= a a)))
```

```
> (match-best-fs s2 s3)
(1.2 () ()
 ((= g g) (= g g) (= c c) (= a a)
 (= g g) (= c c) (= t t) (= a a)
 (= c c) (= t t) (= c c) (i g)
 (= a a) (= c c) (= g g) (= a a)
 (= c c) (= t t) (= a a) (o g)
 (= a a)))
```


But, We Forgot Something

- Dynamic Programming Principle:
  - If two partial solutions lead to the same search state, keep only the better one
  - We used memoization earlier
  - Current implementation fails to do this!

When merging queues

1. Drop from **`new`** any state for which
   - There is already a memoized state matching the same tails, and
   - The memoized state has a better score
2. Memoize the remaining new states
3. Drop from **`old`** any state for which
   - The state now memoized has a better score
     (Could only happen if a new state is better.)

σ_1 → σ_2, σ_3 → σ_4

Code for new merging rule

```
(define (my-merge new old)
  (let* ((new-useful
          (filter
           (lambda (n)
             (let ((earlier (table2-get past (ss-x n) (ss-y n))))
                  (cond ((or (not earlier) (better? n earlier))
                         (table2-set! past (ss-x n) (ss-y n) n)
                         #t)
                        (else #f)))))
           new))
         (old-useful
          (filter
           (lambda (o)
             (let ((reached (table2-get past (ss-x o) (ss-y o))))
               (not (better? reached o))))
           old)))
    (priority-merge new-useful old-useful)))
```


Final match-best-fs

```
(define (match-best-fs x y)
  (let ((past (make-table2))
        (start (make-search-state 0.0 x y '())))
    (define (my-merge new old)
      (let* ((new-useful ...)
             (old-useful ...))
       (priority-merge new-useful old-useful)))
    (search start completed? successors my-merge)))
```


Beam Search

```
(define (match-best-fs x y width)
  (let ((past (make-table2))
        (start (make-search-state 0.0 x y '())))
    (define (my-merge new old)
      (let* ((new-useful ...)
             (old-useful ...))
       (list-head!
         (priority-merge new-useful old-useful)))
        width)
    (search start completed? successors my-merge)))
```

```
(define (list-head! lst n)
  (let ((lroot (cons '() lst)))
    (define (iter l i)
      (if (or (zero? i) (null? (cdr l)))
          (set-cdr! l '())
          (iter (cdr l) (- i 1))))
    (iter lroot n)
    (cdr lroot)))
```

Performance

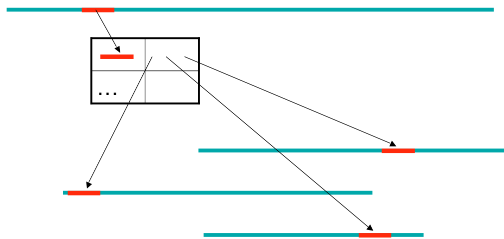
```
(define s1 '(a a t c t g c c t a t t g t c g a c g c))
(define s2 '(a a t c a g c a g c t c a t c g a c g g))
(define s3 '(a g a t c a g c a c t c a t c g a c g g))
```

	Best-First-Search			Beam Search	
	penalty	max queue	# past	min beam	# past
s1,s2	1.8	19	68	1	34
s1,s3	2.8	31	127	9	112
s2,s3	1.2	12	41	6	41
s1s1,s2s2	3.6	57	282	1	68
s1s1,s3s3	5.6	63	486	41	472
s1s1s1s1, s2s2s2s2	7.2	119	1113	1	136
s1s1s1s1, s3s3s3s3	11.2	128	1931	106	1922

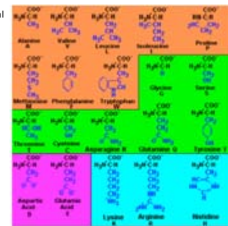
Greatest Improvements

- Good matches will contain large identical subsequences
- Pre-compute table of all occurrences of specific patterns
- Extend match outward (both directions) from these exact matches

BLAST: Find common, extend



http://www.people.virginia.edu/~rjh9u/aminacid.html



Generalize

- DNA
  - Nucleotides: A, C, T, G
  - Mutation rates
  - Insertion/omission penalties
- Proteins
  - Amino Acids: val, leu, ile, met, phe, asn, glu, gln, …
  - Mutation rates
  - Insertion/omission penalties