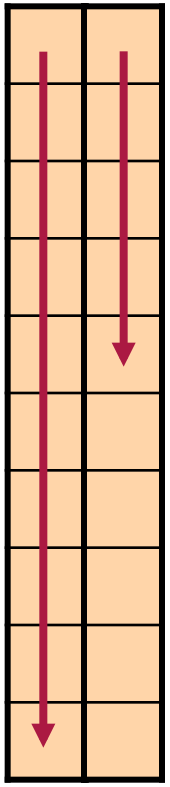
Linear-Space Alignment



Subsequences and Substrings

<u>Definition</u> A string x' is a ***substring*** of a string x,
if x = ux'v for some prefix string u and suffix string v

(similarly, $x' = x_i \ldots x_j$, for some $1 \le i \le j \le |x|$)

A string x' is a ***subsequence*** of a string x
if x' can be obtained from x by deleting 0 or more letters

($x' = x_{i1} \ldots x_{ik}$, for some $1 \le i_1 \le \ldots \le i_k \le |x|$)

<u>Note:</u> a substring is always a subsequence

<u>Example:</u> **x = abracadabra**
y = cadabr; ***substring***
z = brcdbr; ***subseqence, not substring***

Hirschberg's algortihm

Given a set of strings x, y,…, a ***common subsequence*** is a string u that is a subsequence of all strings x, y, …

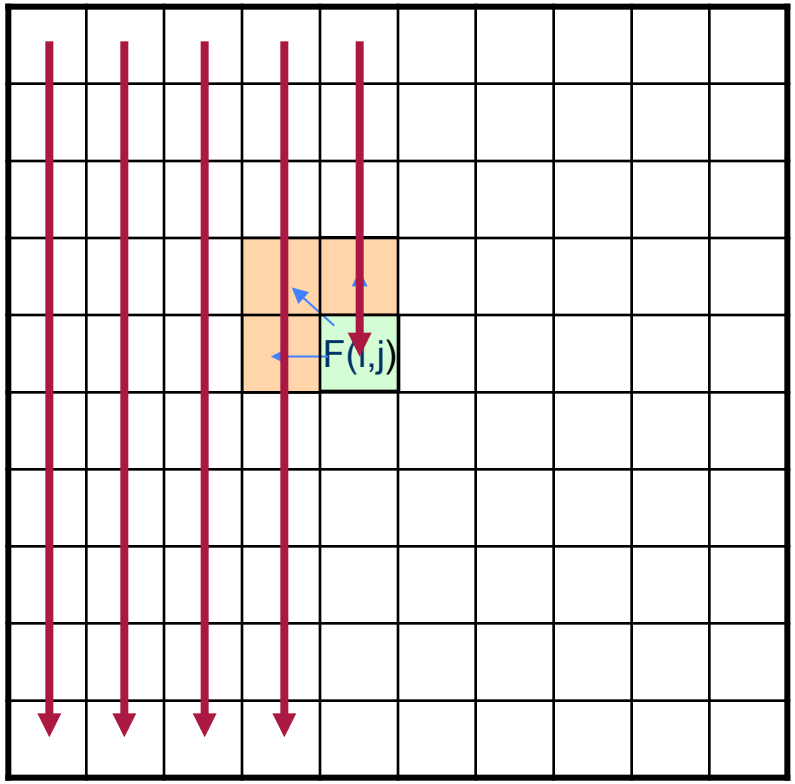
- Longest common subsequence
  - Given strings $x = x_1\ x_2 \ldots x_M$, $y = y_1\ y_2 \ldots y_N$,
  - Find longest common subsequence $u = u_1 \ldots u_k$

- Algorithm:
  - $$F(i, j) = \max \begin{cases} F(i-1, j) \\ F(i, j-1) \\ F(i-1, j-1) + [1, \text{ if } x_i = y_j;\ 0 \text{ otherwise}] \end{cases}$$
  - Ptr(i, j) = *(same as in N-W)*
  - <u>Termination:</u> trace back from Ptr(M, N), and prepend a letter to u whenever
    - Ptr(i, j) = DIAG **<u>and</u>** F(i – 1, j – 1) < F(i, j)

- Hirschberg's original algorithm solves this in linear space

Introduction: Compute optimal score

It is easy to compute F(M, N) in linear space




```
Allocate ( column[1] )
Allocate ( column[2] )

For    i = 1….M
    If      i > 1, then:
            Free( column[ i – 2 ] )
            Allocate( column[ i ] )
    For   j = 1…N
            F(i, j) = …
```

Linear-space alignment

To compute both the optimal score **and** the optimal alignment:

Divide & Conquer approach:

Notation:

x^r, y^r: reverse of x, y

E.g. x = accgg;

x^r = ggcca

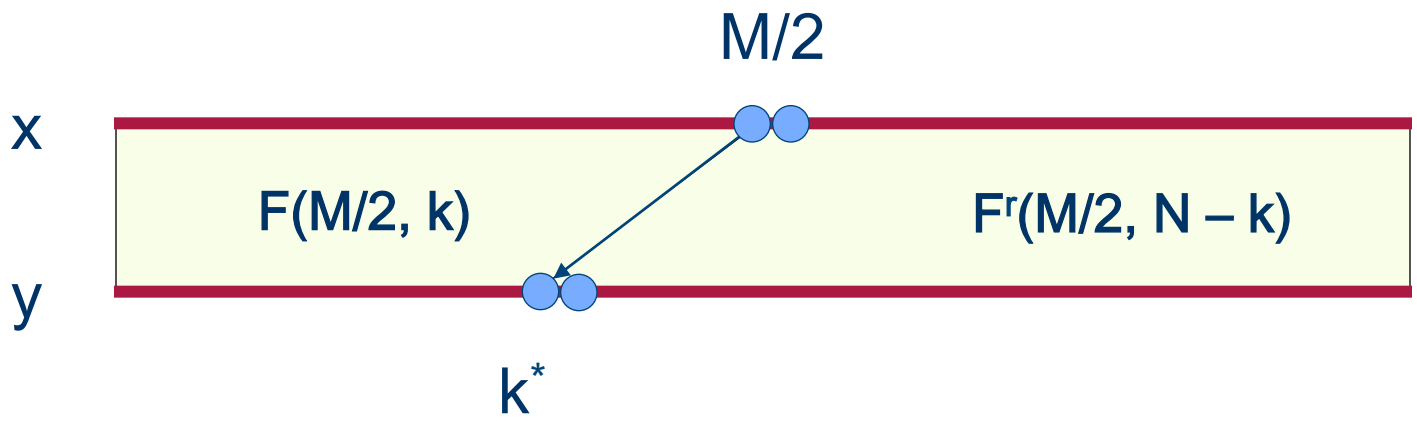
$F^r(i, j)$: optimal score of aligning $x^r_1 \ldots x^r_i$ & $y^r_1 \ldots y^r_j$

same as aligning $x_{M-i+1} \ldots x_M$ & $y_{N-j+1} \ldots y_N$

Linear-space alignment

Lemma: (assume M is even)

$$F(M, N) = \max_{k=0\ldots N}(F(M/2, k) + F^r(M/2, N - k))$$



Example:

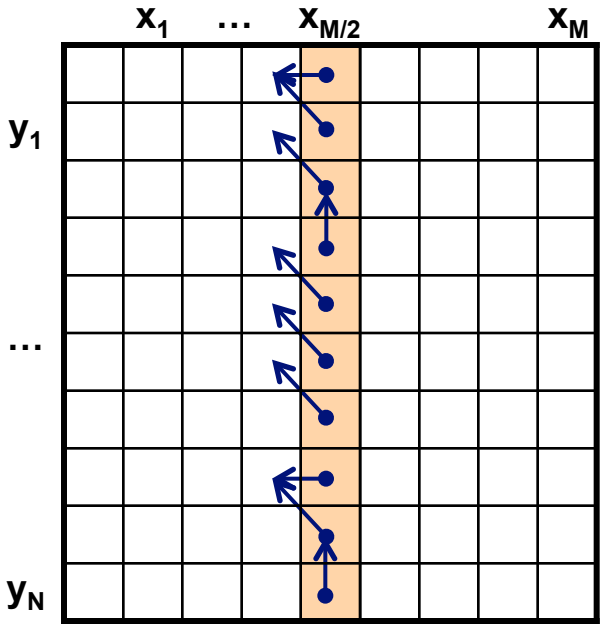
```
ACC-GGTGCCCAGGACTG--CAT
ACCAGGTG----GGACTGGGCAG
```

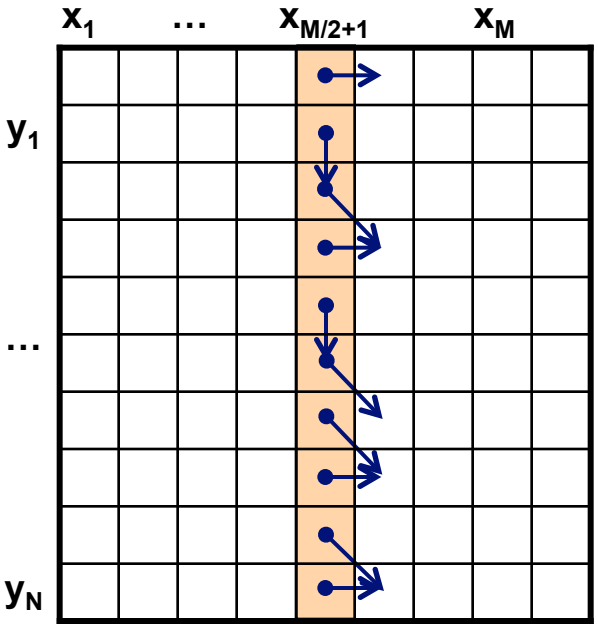
$k^* = 8$

Linear-space alignment

- Now, using 2 columns of space, we can compute for k = 1…M, F(M/2, k), F^r(M/2, N – k)

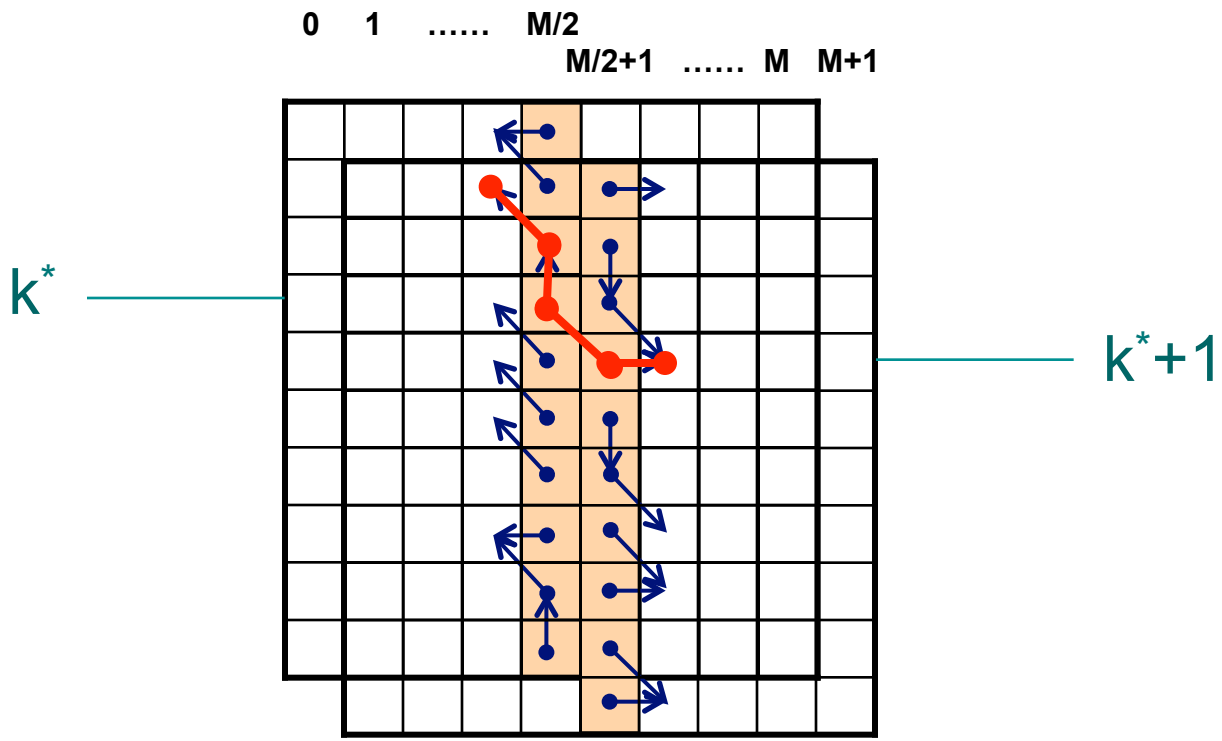
PLUS the backpointers





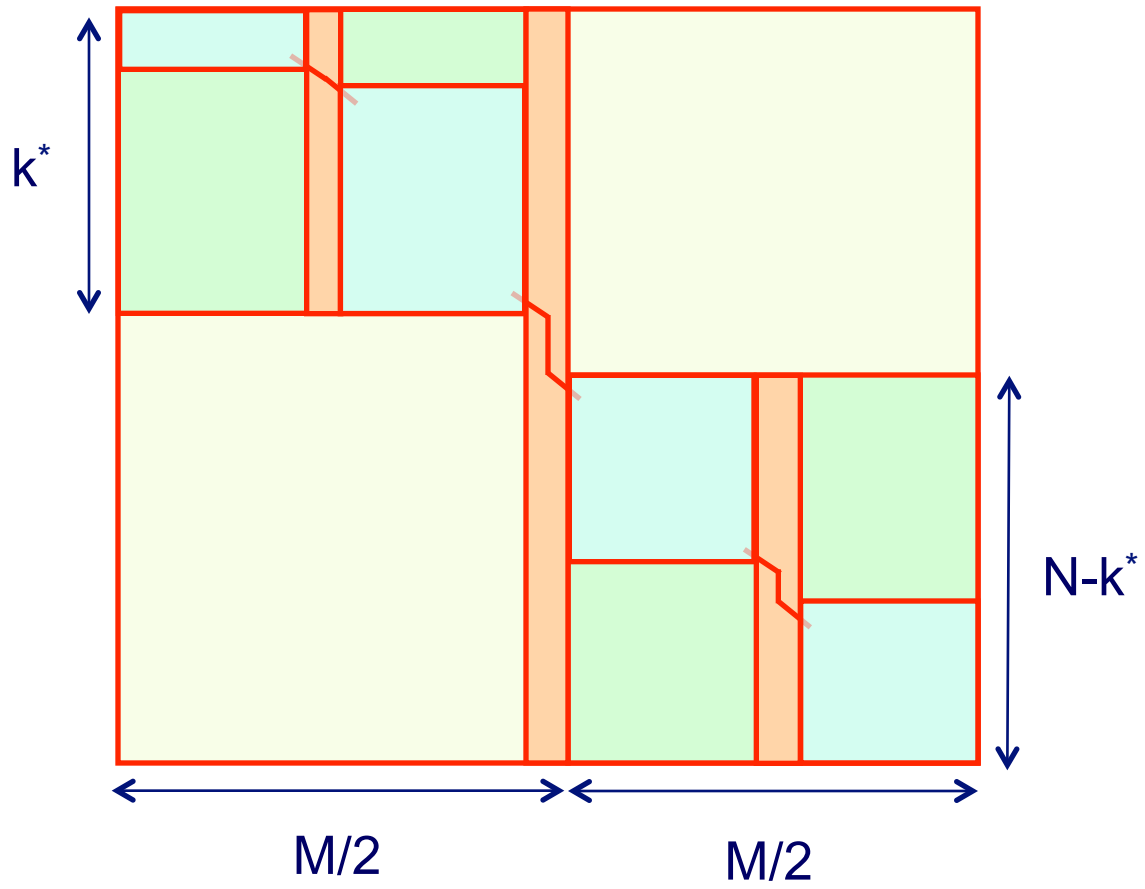
Linear-space alignment

- Now, we can find k^* maximizing $F(M/2, k) + F^r(M/2, N-k)$
- Also, we can trace the path exiting column M/2 from k^*



Linear-space alignment

- Iterate this procedure to the left and right!



Linear-space alignment

<u>Hirschberg's Linear-space algorithm:</u>

MEMALIGN(l, l', r, r'): (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)

1. Let $h = \lceil (l'-l)/2 \rceil$
2. Find (in Time $O((l' - l) \times (r' - r))$, Space $O(r' - r)$)
   the optimal path, L_h, entering column h – 1, exiting column h
   Let k_1 = pos'n at column h – 2 where L_h enters
   k_2 = pos'n at column h + 1 where L_h exits
3. MEMALIGN(l, h – 2, r, k_1)
4. Output L_h
5. MEMALIGN(h + 1, l', k_2, r')

Top level call: MEMALIGN(1, M, 1, N)

Linear-space alignment

Time, Space analysis of Hirschberg's algorithm:

To compute optimal path at middle column,

For box of size $M \times N$,

Space: 2N

Time: cMN, for some constant c

Then, left, right calls cost $c(M/2 \times k^* + M/2 \times (N - k^*)) = cMN/2$

All recursive calls cost

Total Time: $cMN + cMN/2 + cMN/4 + \ldots = 2cMN = O(MN)$

Total Space: O(N) for computation,

O(N + M) to store the optimal alignment

Heuristic Local Aligners

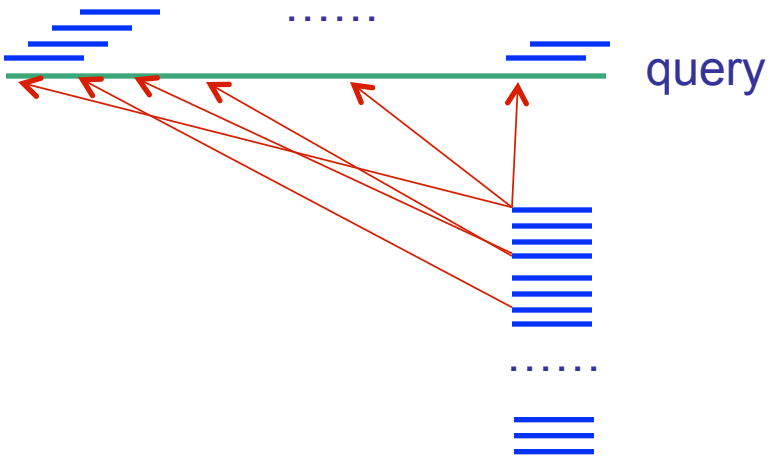
1. The basic indexing & extension technique
2. Indexing: techniques to improve sensitivity
   Pairs of Words, Patterns
3. Systems for local alignment

Indexing-based local alignment

Dictionary:

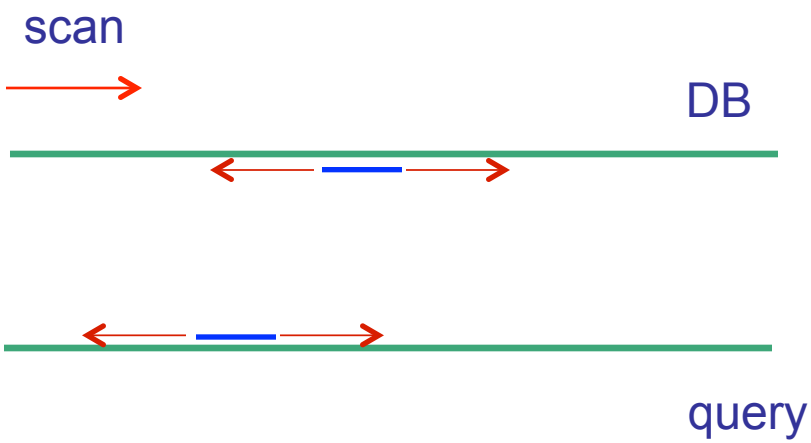
All words of length k (~10)

Alignment initiated between
words of alignment score ≥ T
(typically T = k)



Alignment:

Ungapped extensions until score
below statistical threshold



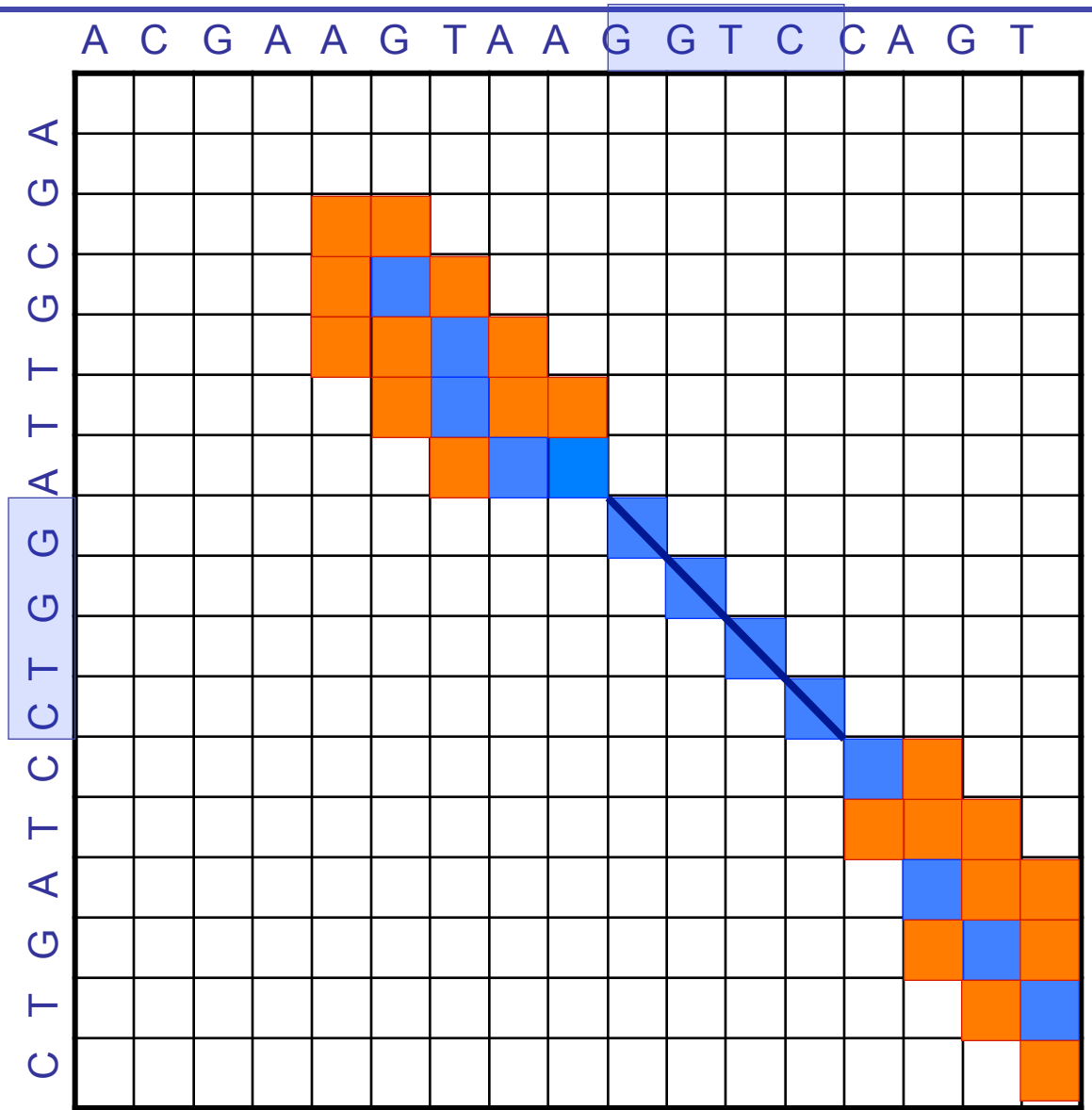
Output:

All local alignments with score
> statistical threshold

Indexing-based local alignment—Extensions

<u>Gapped extensions until threshold</u>

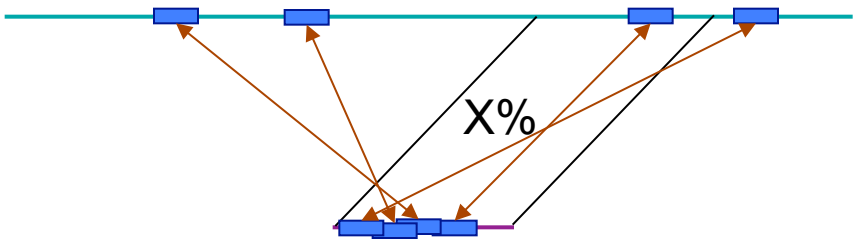
- Extensions with gaps until score < C below best score so far



<u>Output:</u>

```
GTAAGGTCCAGT
GTTAGGTC-AGT
```

Sensitivity-Speed Tradeoff



	long words (k = 15)	short words (k = 7)
Sensitivity		✓
Speed	✓	

Table 3. Sensitivity and Specificity of Single Perfect Nucleotide K-mer Matches as a Search Criterion

	7	8	9	10	11	12	13	14
A. 81%	0.974	0.915	0.833	0.726	0.607	0.486	0.373	0.314
83%	0.988	0.953	0.897	0.815	0.711	0.595	0.478	0.415
85%	0.996	0.978	0.945	0.888	0.808	0.707	0.594	0.532
87%	0.999	0.992	0.975	0.942	0.888	0.811	0.714	0.659
89%	1.000	0.998	0.991	0.976	0.946	0.897	0.824	0.782
91%	1.000	1.000	0.998	0.993	0.981	0.956	0.912	0.886
93%	1.000	1.000	1.000	0.999	0.995	0.987	0.968	0.957
95%	1.000	1.000	1.000	1.000	0.999	0.998	0.994	0.991
97%	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.999
B. K	7	8	9	10	11	12	13	14
F	1.3e+07	2.9e+06	635783	143051	32512	7451	1719	399

(*A*) Columns are for K sizes of 7–14. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated from equation 3 assuming a homologous region of 100 bases. The larger the value of K, the fewer homologies are detected.
(*B*) K represents the size of the perfect match. F shows how many perfect matches of this size expected to occur by chance according to equation 4 in a genome of 3 billion bases using a query of 500 bases.

Sens.

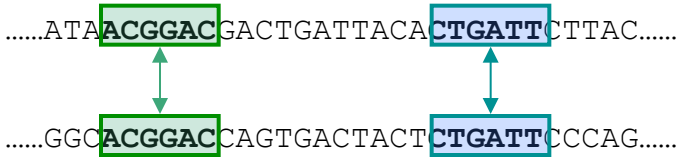
Speed

Kent WJ, Genome Research 2002

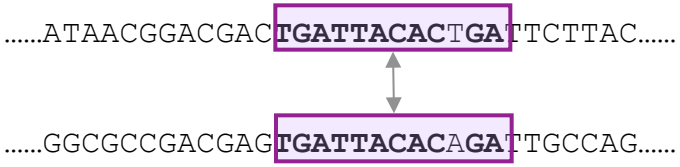
Sensitivity-Speed Tradeoff

Methods to improve sensitivity/speed

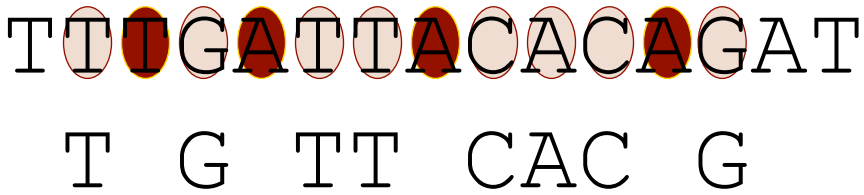
1. Using pairs of words



2. Using inexact words



3. Patterns—non consecutive positions



Measured improvement

Table 7. Sensitivity and Specificity of Multiple (2 and 3) Perfect Nucleotide K-mer Matches as a Search Criterion

	2,8	2,9	2,10	2,11	2,12	3,8	3,9	3,10	3,11	3,12
A. 81%	0.681	0.508	0.348	0.220	0.129	0.389	0.221	0.112	0.051	0.021
83%	0.790	0.638	0.475	0.326	0.208	0.529	0.339	0.193	0.099	0.045
85%	0.879	0.762	0.615	0.460	0.318	0.676	0.487	0.313	0.180	0.093
87%	0.942	0.866	0.752	0.611	0.461	0.809	0.649	0.470	0.305	0.177
89%	0.978	0.940	0.868	0.761	0.625	0.910	0.801	0.648	0.476	0.314
91%	0.994	0.980	0.947	0.884	0.787	0.969	0.914	0.815	0.673	0.505
93%	0.999	0.996	0.986	0.962	0.912	0.993	0.976	0.933	0.851	0.722
95%	1.000	1.000	0.998	0.993	0.979	0.999	0.997	0.987	0.961	0.902
97%	1.000	1.000	1.000	1.000	0.999	1.000	1.000	0.999	0.997	0.987
B. N,K	2,8	2,9	2,10	2,11	2,12	3,8	3,9	3,10	3,11	3,12
F	524	27	1.4	0.1	0.0	0.1	0.0	0.0	0.0	0.0

(*A*) Columns are for N sizes of 2 and 3 and K sizes of 8–12. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated by equation 10. (*B*) N and K represent the number and size of the near-perfect matches, respectively. F shows how many perfect clustered matches expected to occur by chance according to equation 14 in a translated genome of 3 billion bases using a query of 167 amino acids.

Table 5. Sensitivity and Specificity of Single Near-Perfect (One Mismatch Allowed) Nucleotide K-mer Matches as a Search Criterion

	12	13	14	15	16	17	18	19	20	21	22
A. 81%	0.945	0.880	0.831	0.721	0.657	0.526	0.465	0.408	0.356	0.255	0.218
83%	0.975	0.936	0.904	0.820	0.770	0.649	0.591	0.535	0.480	0.361	0.318
85%	0.991	0.971	0.954	0.900	0.865	0.767	0.719	0.669	0.619	0.490	0.445
87%	0.997	0.990	0.983	0.954	0.935	0.867	0.833	0.796	0.757	0.634	0.591
89%	1.000	0.997	0.995	0.984	0.976	0.939	0.920	0.897	0.872	0.775	0.741
91%	1.000	1.000	0.999	0.996	0.994	0.979	0.971	0.962	0.950	0.890	0.869
93%	1.000	1.000	1.000	0.999	0.999	0.996	0.994	0.991	0.988	0.963	0.954
95%	1.000	1.000	1.000	1.000	1.000	1.000	0.999	0.999	0.999	0.994	0.992
97%	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
B. K	12	13	14	15	16	17	18	19	20	21	22
F	275671	68775	17163	4284	1070	267	67	17	4.2	1.0	0.3

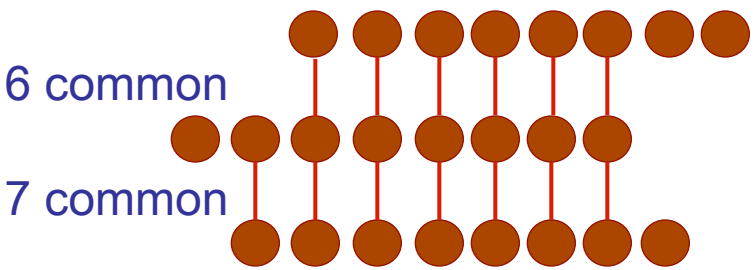
(*A*) Columns are for K sizes of 12–22. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated by equation 6 assuming a homologous region of 100 bases. (*B*) K represents the size of the near-perfect match. F shows how many perfect matches of this size expected to occur by chance accordin[illegible] billion bases using a query of 500 bases.

Kent WJ, Genome Research 2002

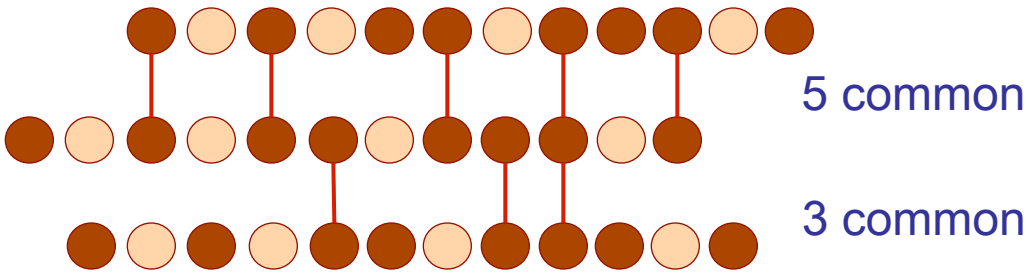
Non-consecutive words—Patterns

Patterns increase the likelihood of *at least one* match within a long conserved region

Consecutive Positions



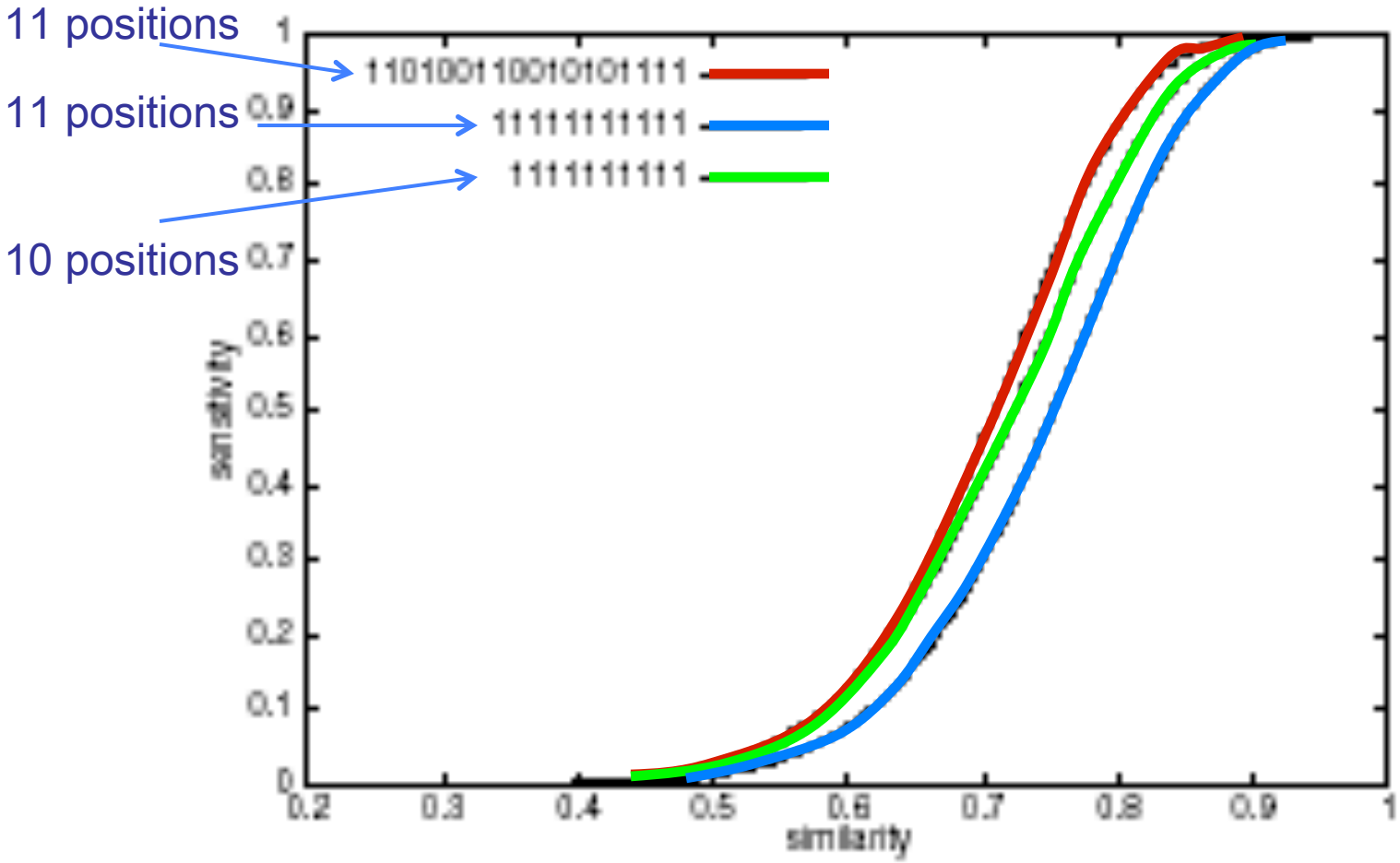
Non-Consecutive Positions



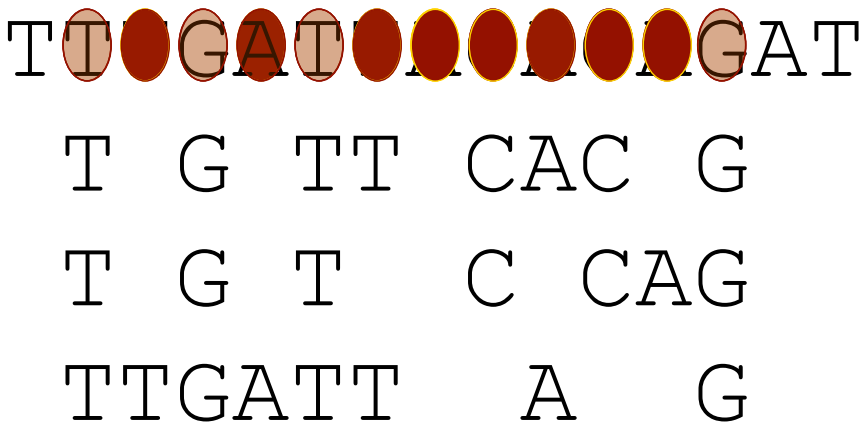
On a 100-long 70% conserved region:

	Consecutive	Non-consecutive
Expected # hits:	1.07	0.97
Prob[at least one hit]:	0.30	0.47

Advantage of Patterns



Multiple patterns



How long does it take to search the query?

Seed	Pattern	Pr[detection]	Alignments Found	Time (s)
π_c	$\{0,1,2,3,4,5,6,7,8,9,10\}$	0.600	66419	15802
π_{c10}	$\{0,1,2,3,4,5,6,7,8,9\}$	0.707	73539	24129
π_{ph}	$\{0,1,2,4,7,9,12,13,15,16,17\}$	0.691	75518	16717
π_{N_0}	$\{0,1,2,4,7,8,11,13,16,17,18\}$	0.683	75231	16225
π_{N_5}	$\{0,1,2,3,5,6,7,10,12,13,14\}$	0.709	75547	16817
$\pi_1 + \pi_2$	$\{0,1,2,4,5,9,14,16,17,18,19,20\}+$ $\{0,1,2,3,4,6,7,8,10,11,12,13\}$	0.744	77211	22033

Buhler et al. RECOMB 2003
Sun & Buhler RECOMB 2004

Human Genome Resequencing

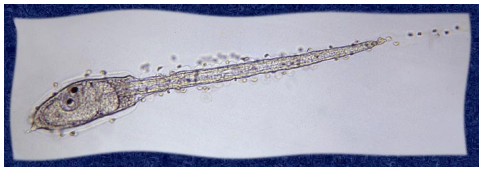
Which human did we sequence?

Answer one:

Answer two: “it doesn’t matter”

Polymorphism rate: number of letter changes between two different members of a species

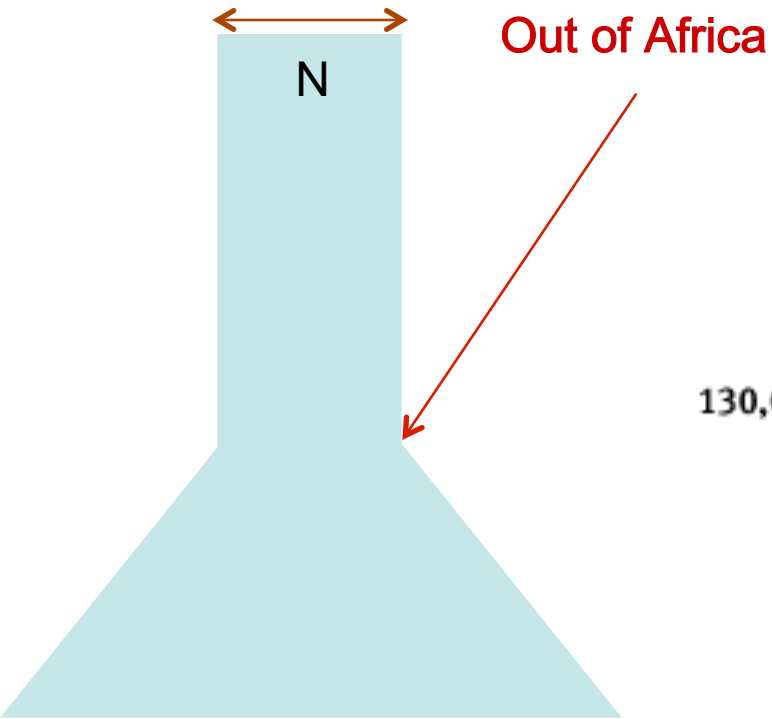
Humans: ~1/1,000



Other organisms have much higher polymorphism rates

- Population size!

Why humans are so similar



Heterozygosity: H

$H = 4Nu/(1 + 4Nu)$

$u \sim 10^{-8}$, $N \sim 10^{4}$

$\Rightarrow H \sim 4\times10^{-4}$

A small population that interbred reduced the genetic variation

Out of Africa ~ 40,000 years ago

DNA Sequencing

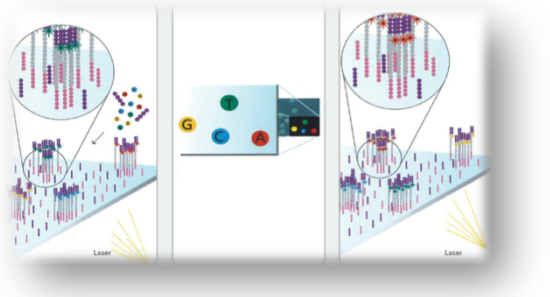
Goal:

Find the complete sequence of A, C, G, T's in DNA

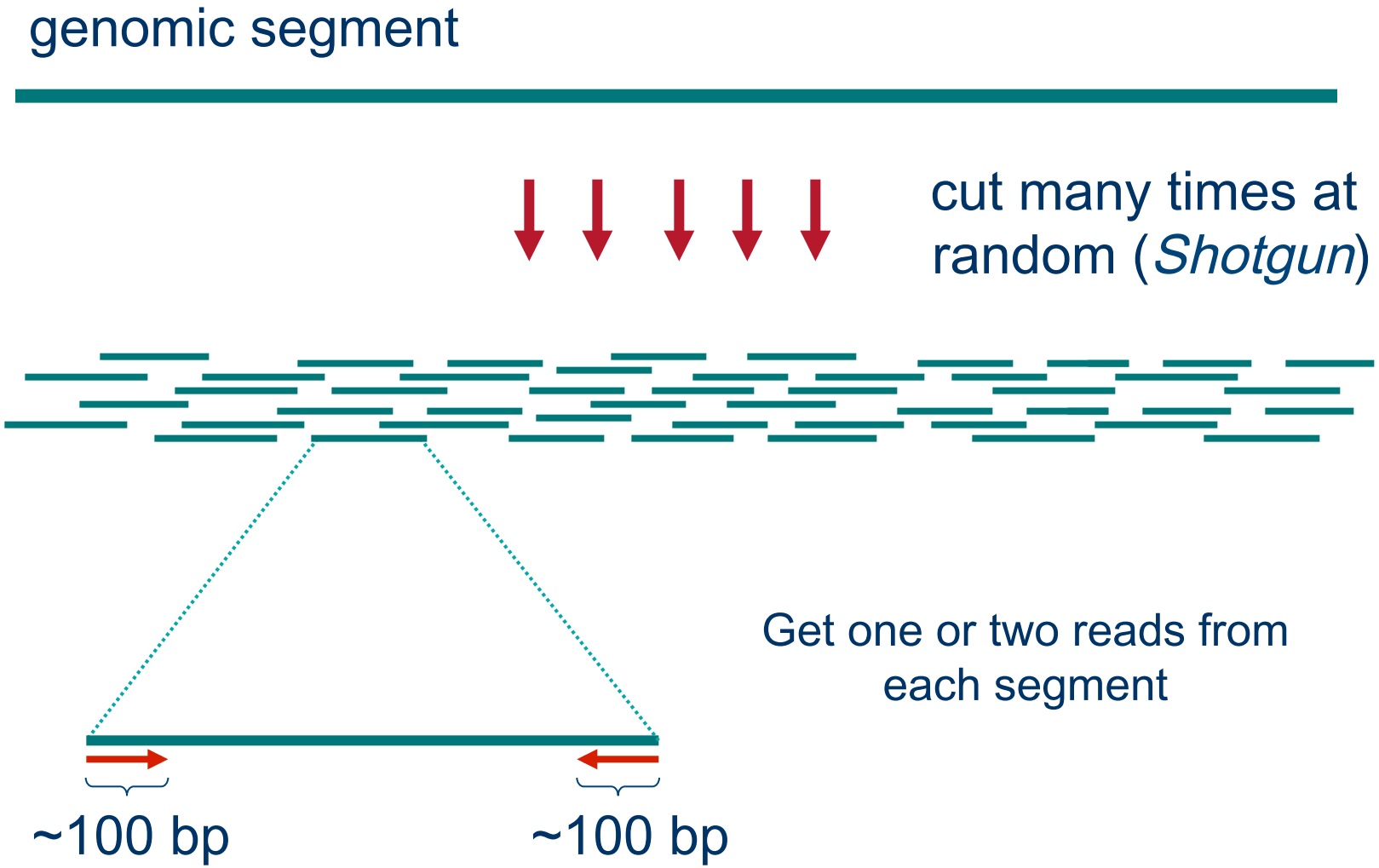
Challenge:

There is no machine that takes long DNA as an input, and gives the complete sequence as output

Can only sequence ~150 letters at a time



Method to sequence longer regions



Definition of Coverage

Length of genomic segment: **G**
Number of reads: **N**
Length of each read: **L**

<u>Definition:</u> Coverage **C = N L / G**

How much coverage is enough?

Lander-Waterman model: **Prob[not covered bp] = e^{-C}**
Assuming uniform distribution of reads, C=10 results in 1 gapped region /1,000,000 nucleotides

Repeats

Bacterial genomes: 5%
Mammals: 50%

Repeat types:

- **Low-Complexity DNA** (e.g. ATATATATACATA…)
- **Microsatellite repeats** $(a_1 \ldots a_k)^N$ where k ~ 3-6
  (e.g. CAGCAGTAGCAGCACCAG)
- **Transposons**
  - **SINE** (Short Interspersed Nuclear Elements)
    e.g., ALU: ~300-long, 10^6 copies
  - **LINE** (Long Interspersed Nuclear Elements)
    ~4000-long, 200,000 copies
  - **LTR retroposons** (Long Terminal Repeats (~700 bp) at each end)
    cousins of HIV
- **Gene Families** genes duplicate & then diverge (paralogs)
- **Recent duplications** ~100,000-long, very similar copies

Two main assembly problems

- De Novo Assembly

- Resequencing

Human Genome Variation

SNP

TGCTGAGA
TGCCGAGA

Novel Sequence

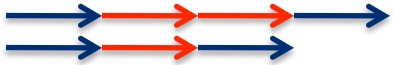
TGCTCGGAGA
TGC - - - GAGA

Inversion

Mobile Element or
Pseudogene Insertion

Translocation

Tandem Duplication



Microdeletion

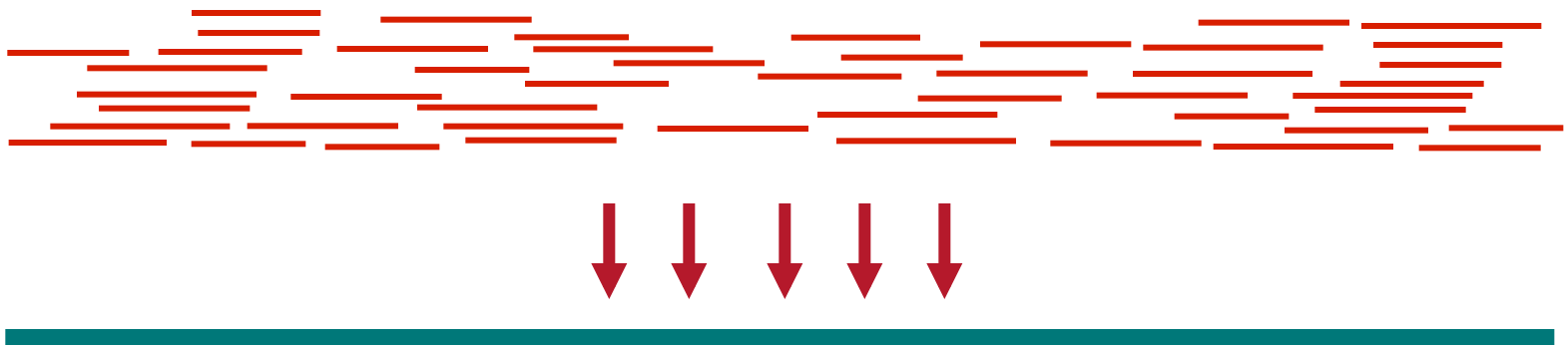
TGC - - AGA
TGCCGAGA

Transposition

Large Deletion

Novel Sequence
at Breakpoint

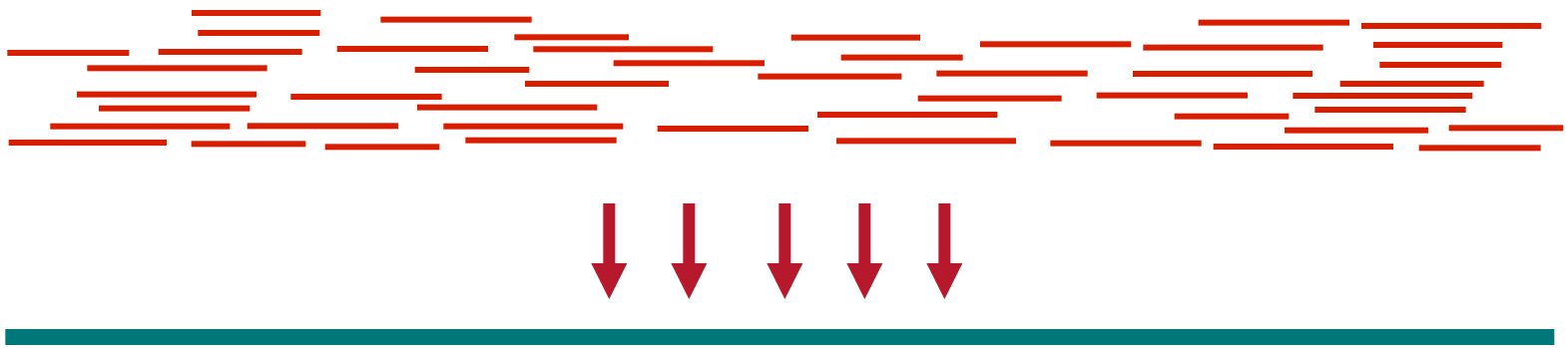
Read Mapping



```
                CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
            TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
        GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
    GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
.....AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Want ultra fast, highly similar alignment
- Detection of genomic variation

Read Mapping – Burrows-Wheeler Transform



```
                CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
            TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
        GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
    GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
.....AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Modern fast read aligners: BWT, Bowtie, SOAP
  - Based on ***Burrows-Wheeler transform***

Burrows-Wheeler Transform

ANA

X = **BANANA**

BANANA	BANANA
ANANA	ANANA
NANA	NANA
ANA	ANA
NA	NA
A	A

suffixes of BANANA

Burrows-Wheeler Transform

ANA

X = BANANA$

BANANA$	BANANA$
ANANA$	ANANA$
NANA$	NANA$
ANA$	ANA$
NA$	NA$
A$	A$
$	$

Burrows-Wheeler Transform

ANA

X = BANANA$

BANANA$	BANANA$	BANANA$
ANANA$B	ANANA$B	ANANA$B
NANA$BA	NANA$BA	NANA$BA
ANA$BAN	ANA$BAN	ANA$BAN
NA$BANA	NA$BANA	NA$BANA
A$BANAN	A$BANAN	A$BANAN
$BANANA	$BANANA	$BANANA

Burrows-Wheeler Transform

ANA

X = BANANA$

BANANA$	BANANA$	$BANANA
ANANA$B	ANANA$B	A$BANAN
NANA$BA	NANA$BA	ANA$BAN
ANA$BAN	ANA$BAN	ANANA$B
NA$BANA	NA$BANA	BANANA$
A$BANAN	A$BANAN	NA$BANA
$BANANA	$BANANA	NANA$BA

Burrows-Wheeler Transform

ANA

X = BANANA$

BANANA$	BANANA$	$BANANA
ANANA$B	ANANA$B	A$BANAN
NANA$BA	NANA$BA	ANA$BAN
ANA$BAN	ANA$BAN	ANANA$B
NA$BANA	NA$BANA	BANANA$
A$BANAN	A$BANAN	NA$BANA
$BANANA	$BANANA	NANA$BA

Burrows-Wheeler Transform

ANA

X = BANANA$

BANANA$	BANANA$	$BANANA
ANANA$B	ANANA$B	A$BANAN
NANA$BA	NANA$BA	ANA$BAN
ANA$BAN	ANA$BAN	ANANA$B
NA$BANA	NA$BANA	BANANA$
A$BANAN	A$BANAN	NA$BANA
		NANA$BA

BWT matrix of string ‘BANANA’

BWT(BANANA) = ANNB$AA

Suffix Arrays

$BANANA	1	$BANANA
A$BANAN	2	A$BANAN
ANA$BAN	3	ANA$BAN
ANANA$B	4	ANANA$B
BANANA$	5	BANANA$
NA$BANA	6	NA$BANA
NANA$BA	7	NANA$BA

Suffixes are sorted in the BWT matrix

S(i) = j, where $X_j \ldots X_n$ is the i-th suffix lexicographically

	1	2	3	4	5	6	7
X	B	A	N	A	N	A	$
S	7	6	4	2	1	5	3
BWT(X)	A	N	N	B	$	A	A

BWT(X) constructed from S:
At each position, take the letter to the left of the one pointed by S

Reconstructing BANANA

$BANANA	A	$	A$	$B	A$B	$BA
A$BANAN	N	A	NA	A$	NA$	A$B
ANA$BAN	N	A	NA	AN	NAN	ANA
ANANA$B	B	A	BA	AN	BAN	ANA
BANANA$	$	B	$B	BA	$BA	BAN
NA$BANA	A	N	AN	NA	ANA	NA$
NANA$BA	A	N	AN	NA	ANA	NAN

BWT matrix of string ‘BANANA’

sort → append BWT → sort → append BWT → sort

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

<u>Lemma.</u> The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

<u>Lemma.</u> The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

```
A$BANAN
NA$BANA
NANA$BA
BANANA$
$BANANA
ANA$BAN
ANANA$B
```

Reconstructing BANANA - faster

<u>Lemma.</u> The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string 'BANANA'

```
A$BANAN
NA$BANA
NANA$BA
BANANA$
$BANANA
ANA$BAN
ANANA$B
```

```
A$BANAN
ANA$BAN
ANANA$B
```

Same words, same sorted order

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

<u>Lemma.</u> The i-th occurrence of character ‘a’ in last column is the same text character as the i-th occurrence of ‘a’ in the first column

LF(): Map the i-th occurrence of character ‘a’ in last column to the first column

LF(r): Let row r contain the i-th occurrence of ‘a’ in last column

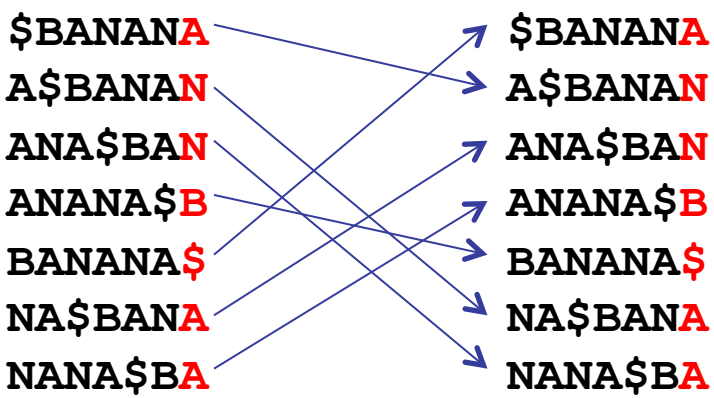
Then, LF(r) = r’; r’: i-th row starting with ‘a’

Reconstructing BANANA - faster

LF(r): Let row r be the i-th occurrence of ‘a’ in last column
Then, LF(r) = r’; r’: i-th row starting with ‘a’

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’



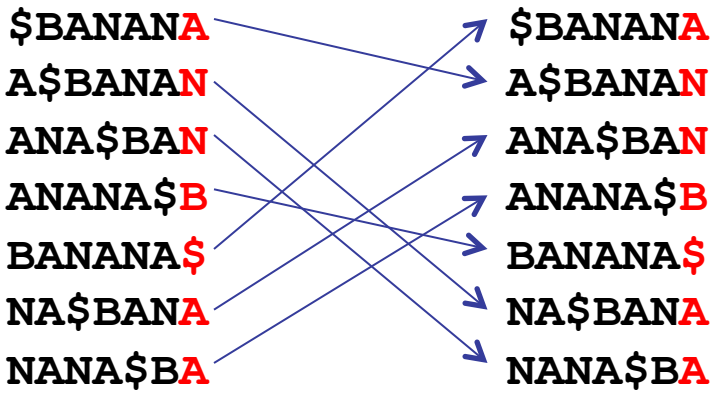
LF[] = [2, 6, 7, 5, 1, 3, 4]

Row LF(r) is obtained by rotating row r one position to the right

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’



LF[] = [2, 6, 7, 5, 1, 3, 4]

Computing LF() is easy:

Let C(a): # of characters smaller than ‘a’

Example: C($) = 0; C(A) = 1; C(B) = 4; C(N) = 5

Let row r end with the i-th occurrence of ‘a’ in last column

Then, LF(r) = C(a) + i (why?)

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

	A	N	N	B	$	A	A	
C()	1	5	5	4	0	1	1	C() copied for convenience
index i	1	1	2	1	1	2	3	indicating this is i-th occurrence of ‘c’
LF()	2	6	7	5	1	3	4	LF() = C() + i

Reconstruct BANANA:

```
S := “”; r := 1; c := BWT[r];
UNTIL c = ‘$’ {
        S := cS;
        r := LF(r);
        c := BWT(r); }
```

Credit: Ben Langmead thesis

Searching for ANA

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

L(W): lowest index in BWT matrix where W is prefix
U(W): highest index in BWT matrix where W is prefix

Example:
L(“NA”) = 6
U(“NA”) = 7

<u>Lemma (prove as exercise)</u>
L(aW) = C(a) + i +1,
where i = # ‘a’s up to L(W) – 1 in BWT(X)
U(aW) = C(a) + j,
where j = # ‘a’s up to U(W) in BWT(X)

Example:
L(“ANA”) = C(‘A’) + # ‘A’s up to (L(“NA”) – 1) + 1
= 1 + (# ‘A’s up to 5) + 1
= 1 + 1 + 1 = 3
U(“ANA”) = 1 + # ‘A’s up to U(“NA”) = 1 + 3 = 4

Searching for ANA

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

```
Let
LFC(r, a) = C(a) + i, where i = #’a’s up to r in BWT

ExactMatch(W[1…k]) {

    a := W[k];
    low := C(a) +1;
    high := C(a+1);    // a+1: lexicographically next char
    i := k – 1;
    while (low <= high && i >= 1) {
            a = W[i];
            low = LFC(low – 1, a) + 1;
            high = LFC(high, a);
            i := i – 1; }
    return (low, high);
}
```

Credit: Ben Langmead thesis

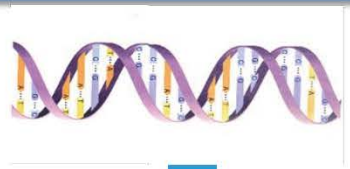
Summary of BWT algorithm

<u>Suffix array of string X:</u>

$S(i) = j$, where $X_j \ldots X_n$ is the j-th suffix lexicographically

- BWT follows immediately from suffix array
  - Suffix array construction possible in O(n), many good O(n log n) algorithms
- Reconstruct X from BWT(X) in time O(n)
- Search for all exact occurrences of W in time O(|W|)
- BWT(X) is easier to compress than X

BWT Index Construction



Reference Sequence Construction

BWT Construction

BWT-auxiliary Structure Construction (C & O arrays) and Compression

.bwt

Reference

TTATTT...ATGTGCCTTTGAAA...GTTAAACCT...AAATT.....AATTT...AGGTTTAAAC..TTTCAAAGGCACAT..AAATAA$

Forward

Reverse Complement

BWT

ACGTTA..TTCTGAATGTGACC…TCCAGACGA…CCATT.....AGTTC…CGGATT AGAT…AAGTACCGTGTGAT…CCAGAT

Compressed BWT (4 bases/byte)

TTATTT...ATGTGCCTT.....$GTTGGTTAATAA

C-array

$:	0
T:	55000
C:	1044814
G:	7814189
A:	1

O-array

	T	G	C	A
0:	1	0	0	0
1:	2	0	0	0
2:	.....			
3:	.....			
..	.....			
|G|-1:	.....			

SA

0:	|G|-1
1:	64
2:	144814
3:	781414689
..:	...
|G|-1:	1484

Memory Consumption

For a genome of length n:

-- occurrence array O(.,.) needs 4nlogn bits

→ sampling: store only O(.,k) for e.g. k = 128

→ use BWT to compute missing counts

-- suffix array SA(.) needs nlogn bits

→ sampling: store SA(k) for e.g. k = 32

→ use inverse compressed suffix array

BWA Inexact Matching

Allow up to *n* mismatches/gaps.

Backwards-search extension:
Given read W, keep track of multiple possible partial alignments of W

Partial alignment 4-tuple: (i, z, L , U)

$I \leftarrow \emptyset$
$I \leftarrow I \cup \textsc{InexRecur}(W, i-1, z-1, k, l)$
for each $b \in \{A, C, G, T\}$ **do**
  $k \leftarrow C(b) + O(b, k-1) + 1$
  $l \leftarrow C(b) + O(b, l)$
  if $k \leq l$ **then**
    $I \leftarrow I \cup \textsc{InexRecur}(W, i, z-1, k, l)$
    if $b = W[i]$ **then**
      $I \leftarrow I \cup \textsc{InexRecur}(W, i-1, z, k, l)$
    else
      $I \leftarrow I \cup \textsc{InexRecur}(W, i-1, z-1, k, l)$

BWA Inexact Matching

```
I ← ∅
I ← I ∪ InexRecur(W, i−1, z−1, k, l)
for each b ∈ {A, C, G, T} do
    k ← C(b) + O(b, k−1) + 1
    l ← C(b) + O(b, l)
    if k ≤ l then
        I ← I ∪ InexRecur(W, i, z−1, k, l)
        if b = W[i] then
            I ← I ∪ InexRecur(W, i−1, z, k, l)
        else
            I ← I ∪ InexRecur(W, i−1, z−1, k, l)
```

W = ACTGTGT

Partial alignment 4-tuple: (i = 4, z = 3, L , U)

Recursive step:

A	C	T	G	gap-ref	gap-read
AGT	CGT	TGT	GGT	∓GT	*GT
z-1	z-1	z	z-1	z-1	z-1
i-1	i-1	i-1	i-1	i-1	i
$L^A U^A$	$L^C U^C$	$L^T U^T$	$L^G U^G$	LU	$L^A U^A$ $L^C U^C$ $L^T U^T$ $L^G U^G$
…GAGT	…GCGT	…GTGT	…GGGT	…G-GT	…GT[A/C/T/G]GT
…GTGT	…GTGT	…GTGT	…GTGT	…GTGT	…GT - GT

L^A=C(A) + O(A, L-1) + 1

U^A=C(A) + O(A, L)

BWA Heuristics

- Lower bound array D, where $D(i)$:= LB on number of differences of exactly matching R[0,i] with the reference (can be computed in O(|R|) time → check $n < D(i)$ instead of $n < 0$)
- Process best partial alignments first: use a *min*-priority heap to store alignment entries (instead of recursion)
- Prune out alignments considered sub-optimal (although they might have fewer than n differences):

  dynamically adjust search parameters (e.g. n):

  (1) stop if # top hits exceeds a threshold (=30),

  (2) set $n = nbest + 1$, where $nbest$ is the # of differences in top hit
- Seeding: limit the number of differences in the *seed* sequence (first k bp)
- Disallow indels at the ends of the read