1-Read-mapping

Sunday, January 11, 2026 1:54 PM

Setup: Given two strings $S = s_1 s_2 \cdots s_m$

$T = t_1 t_2 \cdots t_n$

We want to measure the semi-global alignment of T to S.

Equivalently, consider the modified edit distance to transform $S \to T$ where we are allowed to first delete any prefix and/or suffix of S for free.

Aside: Read mapping & alignment are often used interchangeably now.

Strictly speaking, mapping is just finding the location (i.e. size of prefix deleted) whereas alignment is finding the full set of edits.

Ex S = AACCTTGGGATTACCAGTTATAGGTTCCAA

T = GATTACAGATTACA

```
GATTACCAG-TTATA
|||||||  ||  |||
GATTAC-AGAATTACA
```

← 12 matches

↑ del ↑ ins ↑ sub

3 errors

Recall: Global alignment of two strings via DP (dynamic programming)

Compute a score $F_{i,j}$ for optimal alignment of $S_i = s_1 \cdots s_i$, $T_j = t_1 \cdots t_j$

Recursion

$$F_{i,j} = \max \begin{cases} F_{i-1,j-1} + \text{Score}(s_i, t_j) \\ F_{i,j-1} + G \\ F_{i-1,j} + G \end{cases}$$

ex. $\text{Score}(s_i, t_j) = \mathbb{1}_{s_i = t_j} - \mathbb{1}_{s_i \neq t_j}$ ← positive score for matching char, neg score for mismatches

gap penalty

ex $G = -1$

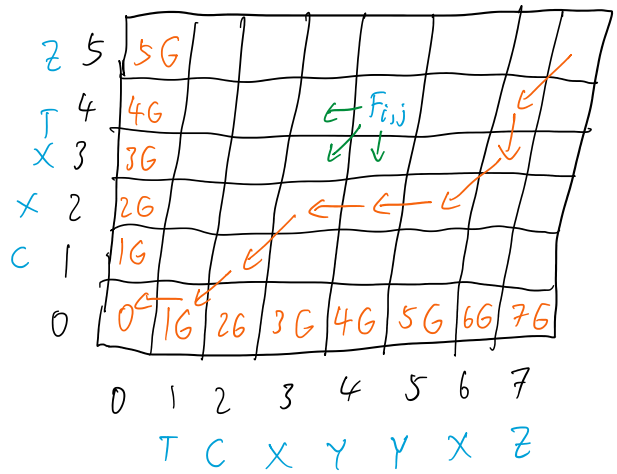
Base case:

$F_{i,0} = i \cdot G$

$F_{0,j} = j \cdot G$

aligns i characters to 0 must use i gaps

DP array:



Backtrack to find actual alignment

move right = add gap to T, char from S

move up = add gap to S, char from T

move diag = choose both chars

Ex $S = TCXYYXZ$ $\quad$ $T C X Y Y X - Z$

$T = CXXTZ$ $\quad$ $- C X - - X T Z$

Generalization: Local alignment of two strings.

Changes: • Set minimum cell to 0 ← can start anywhere

• backtrack from max entry of matrix

Recursion:

$$F_{i,j} = \max \begin{cases} F_{i-1,j-1} + \mathbb{1}_{s_i = t_j} - \mathbb{1}_{s_i \neq t_j} \\ F_{i,j-1} + G \\ F_{i-1,j} + G \\ 0 \end{cases}$$

Base case: $F_{i,0} = 0$

$F_{0,j} = 0$

Backtracking stops when hit 0.

Runtime: $O(1)$ to fill each entry

$O(mn)$ running time.

Read Mapper 1: Do a local alignment of S and T.

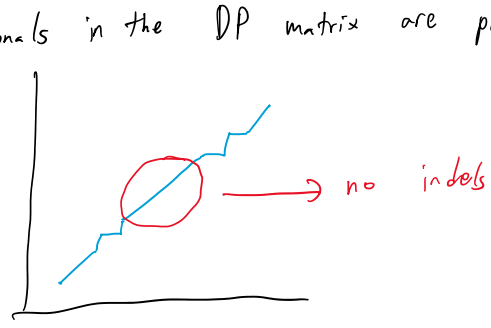
Read Mapper 1: Do a local alignment of S and T.
Highest scoring local alignments will be good read mappings.

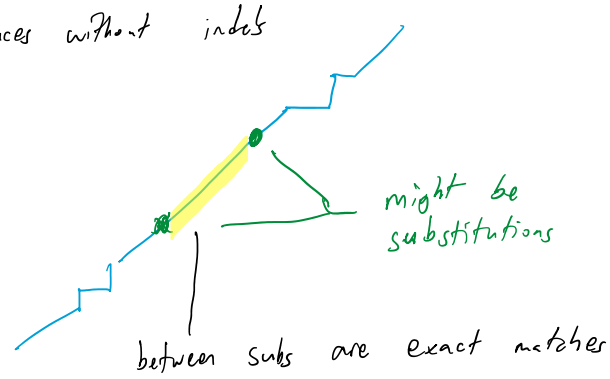
Runtime:
$O(mn)$ DP matrix construction,
$O(mn)$ find highest scoring cell (scan)
$+ \underline{O(n)}$ backtracking ← why?
$O(mn)$ ← too slow!

Can we do things faster by first finding a good location?

Read Mapper 2: Seed and Extend

Notice: diagonals in the DP matrix are places without indels





Idea: First find exact matches, then just align in a local window

Runtime: ? $O(1)$ or $O(n)$ to find exact match ?
$O(n^2)$ to do DP. ← better than $O(mn)$

Dictionary – stores key, value pairs
- add(k, v): adds key k with value v to D
- query(k): returns D[k] = v ← we only need this operation ↳ human reference genome changes slowly
- delete(k): remove key k from D

Static Dictionary implementations
- list ← $O(m)$ look up time

- sorted list ← $O(\log n)$ lookup time
- binary tree } also work for dynamic case
- skip lists } $O(\log n)$ search
- table the size of the universe U. ← $O(|U|)$ memory, but $O(1)$ search
- hash table ← $O(1)$ average case search

Hashing basics

- keys from large universe U. e.g. all possible k-mers Σ^k, where $\Sigma = \{A, C, G, T\}$
- set $S \subseteq U$ of keys we care about ← reusing "S" letter, but this is only a slight abuse of notation as we'll see.

Notice: we can encode k-mers as integers

Recall: A randomized algorithm H for constructing hash functions $h: U \to \{0, 1, \ldots, M-1\}$ is universal if $\forall x \neq y \in U$,

$$\Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{M}.$$

Often, we will also say a set H of hash functions is a universal family of hash functions if the algorithm "choose $h \in H$ uniformly at random" is universal.

Important: By definition, any 2 distinct items must collide w.p. at most $\frac{1}{M}$.

Intuition: If $h: U \to [M]$ maps every $x \in U$ uniformly at random, then collision prob. is exactly $\frac{1}{M}$.

Thm: If H is universal, then $\forall S \subseteq U$ of size N, $\forall x \in U$, if we construct $h \in H$ randomly, then the expected number of collisions b/t x & other elements in S is $\leq \frac{N}{M}$.

Exercise: Prove this for yourself.

Def. A hash table is a dictionary data structure where we use an array A of size M, and a hash function $h: U \to [M]$.
Then we store (k, v) at $A[h(k)]$.

If lucky or if $M \gg m$, then no collisions, so $O(1)$ lookup time.

Collision resolution: keep linked list in each entry of A (separate chaining)

Lookup time for key k = $O(\text{length}(A[h(k)]))$

Lemma: If $h \in H$ is universal, then the expected lookup time is $O(\frac{m}{M})$. e.g. if $M = 2m$, then lookup is $O(1)$.

Exercise Build a read mapper using a hash table of k-mers in a genome.

Complexity: How much space does a hash table of the human genome take?

If $k = 20$, and $M = 2^{32}$, (~ 4 billion, as we have < 3 billion distinct k-mers in human genome)

We need to store a key, which is 20 nucleotides long $\sim$ 40 bits.

We also need to store a location, which is $\log |S| = \log(3 \cdot 10^9) \approx 32$ bits.

$\Rightarrow$ 72 bits = 9 bytes per entry

And 4 billion entries $\Rightarrow$ 36 GB of RAM without counting overhead.

Can we do better?

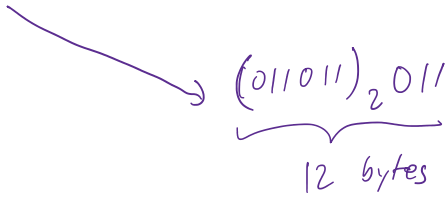
Notice: part of problem is we are storing all the k-mers in the human genome which is very redundant. Can we store it in a compressed fashion?

Compression Basics (lossless compression)

01101101101101 $\longrightarrow$ $(011)_5$

15 bytes 6 bytes

} ignoring issues of how to encode delimeters and ensure decodability



(alternative) still a compression

Run-Length Encoding (one of the simplest compression schemes)

Replaces runs with the character and a repeat number

AAAA GGGG TTTAAAAAAAAAA GGG TTTTTTTT CCC GGG (39 bytes)

↓

A4 G5 T3 A10 G3 T8 C3 G3 (17 bytes)

Works with single-character repeats, but what about longer repeated words?

Ex. Huntintin (HTT) gene encodes glutamine (Q) over and over again.
Wild-type is 6-35 Q. Huntington patients up to 250 Q.

CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAACAGCCGCC

↓ RLE

CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCA2CAGC2GC2

Naive RLE doesn't work.

Lempel-Ziv (LZ77 and LZ78)

CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAACAGCCGCC

↓

(CAG)14CA2CAGC2GC2

or (CAGCAG)7CAACA(GCC)2

} multiple possibilities

Another approach Permute the string

Let's split the string into 3 strings by taking every 3rd char & concat.

CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAACAGCCGCC

↳

C C C C C C C C C C C C C C C C C C C C C C
A A A A A A A A A A A A A A A A A C C
G G G G G G G G G G G G G G G A G G

↳ CCCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAACCGGGGGGGGGGGGGAGG

↳ C18A16C2G14AG2

Problems with permuting

↳ need to store the permutation — e.g. take every 3rd char & concat

↳ permutation needs to be reversible

↳ huge space of possible permutations to explore

↳ O(m log m) space encoding of arbitrary perm, longer than length m string!

Wouldn't it be lovely if there was a single magic permutation that always works?

Burrows-Wheeler Transform (1994)

Recall: Suffix arrays index & sort lists of suffixes

↓ end of string delimiter

(test) T = CAGCAGCAG$

Suffixes:

10 $
9 G$
8 AG$
7 CAG$
6 GCAG$
5 AGCAG$

→

Sorted:

10 $
8 AG$
5 AGCAG$
2 AGCAGCAG$
7 CAG$
4 CAGCAG$

6 GCAG$
5 AGCAG$
4 CAGCAG$
3 GCAGCAG$
2 AGCAGCAG$
1 CAGCAGCAG$

4 CAGCAG$
1 CAGCAGCAG$
9 G$
6 GCAG$
3 GCAGCAG$

Change suffixes to rotations

T = CAGCAGCAG$

~~Suffixes~~: Rotations

10 $CAGCAGCAG
9 G$CAGCAGCA
8 AG$CAGCAGC
7 CAG$CAGCAG
6 GCAG$CAGCA
5 AGCAG$CAG C
4 CAGCAG$CAG
3 GCAGCAG$CA
2 AGCAGCAG$C
1 CAGCAGCAG$

→

Sorted:

10 $CAGCAGCAG
8 AG$CAGCAGC
5 AGCAG$CAGC
2 AGCAGCAG$C
7 CAG$CAGCAG
4 CAGCAG$CAG
1 CAGCAGCAG$
9 G$CAGCAGCA
6 GCAG$CAGCA
3 GCAGCAG$CA

take last column of rotation matrix

BWT: GCCCGG$AAA

This last column is much more compressible.

Why? If there is a repeated pattern CAG, then a string that starts with "AG" will probably end in "C", and we're putting all A-starts (suffixes) together.

Notice: BWT is reversible.

From last col, reconstruct 1st col by sorting.

From first & last col, reconstruct 2nd col by all pairs of letters.

Could even reconstruct suffix array indices by looking for "$"

From first & last col, reconstruct [illegible]

Could even reconstruct suffix array indices by looking for "$"

Define: The T-rank of a letter by how many times it has appeared in the Text

$$T = C_0\ A_0\ G_0\ C_1\ A_1\ G_1\ C_2\ A_2\ G_2\ \$$$

F (first) L (last)

$\$\ C_0\ A_0\ G_0\ C_1\ A_1\ G_1\ C_2\ A_2\ G_2$

$A_2\ G_2\ \$\ C_0\ A_0\ G_0\ C_1\ A_1\ G_1\ C_2$

$A_1\ G_1\ C_2\ A_2\ G_2\ \$\ C_0\ A_0\ G_0\ C_1$

$A_0\ G_0\ C_1\ A_1\ G_1\ C_2\ A_2\ G_2\ \$\ C_0$

$C_2\ A_2\ G_2\ \$\ C_0\ A_0\ G_0\ C_1\ A_1\ G_1$

$C_1\ A_1\ G_1\ C_2\ A_2\ G_2\ \$\ C_0\ A_0\ G_0$

$C_0\ A_0\ G_0\ C_1\ A_1\ G_1\ C_2\ A_2\ G_2\ \$$

$G_2\ \$\ C_0\ A_0\ G_0\ C_1\ A_1\ G_1\ C_2\ A_2$

$G_1\ C_2\ A_2\ G_2\ \$\ C_0\ A_0\ G_0\ C_1\ A_1$

$G_0\ C_1\ A_1\ G_1\ C_2\ A_2\ G_2\ \$\ C_0\ A_0$

Define: The LF-mapping property states that the nth occurrence of a letter in L (the last col) is the same character as the nth occurrence of X in F (the first col)

proof. Consider only rows with X in L. They will be ordered by the prefixes of X.

But now consider only rows with X in R. They will be ordered by the suffixes of X.

By construction, the set of prefixes and suffixes correspond, and uniquely identify which X it is.

Also: It doesn't matter what ranking we give. The rank orders

and uniquely identify which X it is.

Notice: It doesn't matter what ranking we give. The rank orders in L and F will match.

Define: Ascending rank: Assign ranks by order in F & L cols

F(irst) L(ast)

$ C A G C A G C A G$_0$

A$_0$ G $ C A G C A G C$_0$

A$_1$ G C A G $ C A G C$_1$

A$_2$ G C A G C A G $ C$_2$

C$_0$ A G $ C A G C A G$_1$

C$_1$ A G C A G $ C A G$_2$

C$_2$ A G C A G C A G $

G$_0$ $ C A G C A G C A$_0$

G$_1$ C A G $ C A G C A$_1$

→ G$_2$ C A G C A G $ C A$_2$

ascending rank

Jumping around BWM

← Which BWM row starts with G$_2$?

→ So A$_2$ comes before G$_2$

Fast reverse BWT: Start in first row with $ in F

The char X in L is the preceding character.

LF mapping then allows jumping to that occurrence of X in F.

Rinse & Repeat.

Let's turn the BWT into a search index.

FM-index "Full-text Minute-space" by Ferragina & Manzini, FOCS 2000

Basic idea is to store just F & L cols of BWM.

F take 1 integer per alphabet character.

L is compressible.

Then augment with occurrence table to quickly compute ranks.

Query FM-index Let Q = AGC (can't binary search as in suffix array)

Start by looking in F for all C's.

Then use LF to find all G's before C's. (i.e. finding CA)

Build backwards, to get all AGC instances

(if query doesn't exist, will fail. e.g. GCAT)

Need to store ranks: Easy for F. Why? All instances of each letter together.

For L, we build a tally table for every letter.

	F	L	A	C	G	T
	$	G_0	0	0	1	0
	A	C	0	1	1	0
$	A	C	0	2	1	0
A_3	A	C	0	3	1	0
C_3	C	G_1	0	3	2	0
	C	G_2	0	3	3	0
G_3	C	$	0	3	3	0
	G	A	1	3	3	0
	G	A	2	3	3	0
	G	A	3	3	3	0

which G's come before a C?

On G tally table, we go one before the C index, to get 1–3 range

⇒ the 2nd & 3rd G's appear before C's

⇒ G_1 & G_2

O(1) time

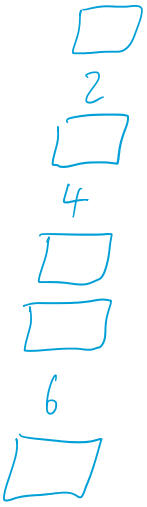
Can save space by only keeping some checkpoints & scanning forward.

Resolving offsets (position)

F	L	SA index	sample
$	G	10	10
A	C	8	8
A	C	5	

we keep every

A	C	5
A	C	
A	C	2
C	G	7
C	G	4
C	$	1
G	A	9
G	A	6
G	A	3



we keep every k th offset, so to find where we are, we just use LF property to walk along at most k times.

FM index components

First col (F): $\sim |\Sigma|$ integers

Last col (L): m characters

SA sample: $m \cdot a$ integers, where a is fraction of rows kept

Tally table: $m \cdot |\Sigma| \cdot b$ integers, where b is fraction of rows checkpointed.

Ex. T = Human genome. $|T| = 3$ billion, $|\Sigma| = 4$, $a = \frac{1}{32}$ $b = \frac{1}{128}$

FM-index stores human genome in ~ 1.5 GB

Full suffix array takes ≥ 12 GB

Hash table ~ 36 GB