

Lecture 1.2

- Algorithm generalities
- Finding exact matches in sequences using suffix arrays
- Hashtables

Algorithms – Some General Remarks

- The most widely used algorithms are the oldest
 - e.g. sorting lists, traversing trees, dynamic programming.
- The challenge in CMB is usually *not* finding *new* algorithms, but rather
- finding *biologically appropriate applications* of old ones.
- Often prefer
 - suboptimal but easy-to-program algorithm over more optimal one
 - or space-efficient algorithm over time-efficient one.
 - *Probabilities* are important in
 - interpreting results
 - guiding search

The most powerful analyses generally involve probabilistic models, rather than deterministic ones.

Genomes are big but computers are fast!

- Typical laptop clock speed: ~ 1 Ghz
 - Potentially billions of CPU instructions / sec
- Important practical consideration in dealing with genome-scale data sets: compared to CPU operations,
 - *non-cache memory accesses* are very slow (100s of cycles)
 - *disk accesses* are even slower (1000s of cycles)
 - for both, random (non-sequential) accesses are much slower than sequential accesses

Exponents & logarithms

- $\log_a(a^b) = b$, $a^{\log_a(b)} = b$ (*log inverts exp*)
- $a^{b+c} = a^b a^c$ $\log_a(df) = \log_a(d) + \log_a(f)$
- $(a^b)^c = a^{bc}$ $\log_a(d^f) = f \log_a(d)$
- $a^0 = 1$ $\log_a(1) = 0$
- $a^1 = a$ $\log_a(a) = 1$
- $a^{-b} = 1 / a^b$ $\log_a(1 / d) = -\log_a(d)$
- $\log_c(b) = \log_a(b) / \log_a(c)$

- $4 = 2^2$
- $4^5 = 2^{10} = 1024 \approx 10^3$
- $4^{10} = 2^{20} \approx 10^6$
- $4^{15} = 2^{30} \approx 10^9$
- $4^n = \# \text{ DNA words of length } n$
- $\log_4(10^9) \approx 15$

Finding perfectly matching subsequences of a sequence

- Idea (*much* more efficient than ‘brute force’ approach):
 - *suffix array* (Manber & Myers, 1990)
 - make list of positions in sequence
 - each position ‘points to’ a *suffix*
 - = subsequence starting at that position & extending to end of sequence
 - lexicographically sort list of pointers
 - process the list: adjacent entries are “maximally agreeing”

Suffix array step 1: List of Pointers to Suffixes

ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC

p_1	ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_2	CCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_3	CTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_4	TGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_5	GCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_6	CACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_7	ACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_8	CTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_9	TAAACCGTACACTGGGTTCAAGAGATTTCCC
p_{10}	AAACCGTACACTGGGTTCAAGAGATTTCCC
p_{11}	AACCGTACACTGGGTTCAAGAGATTTCCC
p_{12}	ACCGTACACTGGGTTCAAGAGATTTCCC

⋮

- The ‘pointers’ are just positions (represented by integers) – *not* (necessarily) memory addresses
- *Do not* store the substrings!

Suffix array step 2:

View as Strings to be Compared

ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC

p_1 ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_2 CCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_3 CTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_4 TGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_5 GCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_6 CACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_7 ACTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_8 CTAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_9 TAAACCGTACACTGGGTTCAAGAGATTTCCC
 p_{10} AAACCGTACACTGGGTTCAAGAGATTTCCC
 p_{11} AACCGTACACTGGGTTCAAGAGATTTCCC
 p_{12} ACCGTACACTGGGTTCAAGAGATTTCCC

⋮

Suffix array step 3:

Sort the Pointers Lexicographically

ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC

p_{10}	AAACCGTACACTGGGTTCAAGAGATTTCCC
p_{11}	AACCGTACACTGGGTTCAAGAGATTTCCC
p_{28}	AAGAGATTTCCC
p_{17}	ACACTGGGTTCAAGAGATTTCCC
p_{12}	ACCGTACACTGGGTTCAAGAGATTTCCC
p_1	ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_7	ACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_{19}	ACTGGGTTCAAGAGATTTCCC
p_{29}	AGAGATTTCCC
p_{31}	AGATTTCCC
p_{33}	ATTTCCC
p_{27}	CAAGAGATTTCCC
	⋮

Finding Matching Subsequences Using the Sorted List of Pointers

- Perfectly matching subsequences
 - (more precisely – the pointers to the starts of those subsequences)are “near” each other in the sorted list
- For a given subsequence, a *longest* perfect match to it is *adjacent* to it in the sorted list
 - (there may be other, equally long matches which are not adjacent, but they are nearby).

(Average Case) Complexity Analysis

- If $N =$ sequence length, sorting can be done with
 - $O(N \log(N))$ comparisons,
 - each requiring $O(\log(N))$ steps on average,for an overall complexity of $O(N(\log(N))^2)$.
 - (Processing the sorted list requires an additional $O(N)$ steps which does not affect the overall complexity).
- Manber & Myers (1990) have more efficient algorithm ($O(N \log(N))$)
- several $O(N)$ algorithms are now known – but the best implementations are not as fast as $O(N \log(N))$ algorithms, even for very large genomes!!
- \exists other, older $O(N)$ methods (‘suffix trees’), but these are
 - much less space efficient,
 - harder to program, and
 - (probably) slower in practice

- Can use to find matches *among multiple sequences* by concatenating them (+ reverse complements)
 - e.g. *sequence assembly* of a large # of reads
- HW #1 asks you to apply this algorithm to find:
 - longest perfectly matching subsequences in 2 genomic sequences & their reverse complements.
- much faster than an $O(N^2)$ algorithm (e.g. Smith-Waterman, or even BLAST), *but*
- limited to finding *exact* matches

Hashtables

- Similar purpose: to store locations of subsequences in a way that allows quick finding of matches
- But using subsequences (or *words*) of a fixed length w
- Idea: work thru the sequence a base at a time.
 - for the word starting at position p :
 - Convert the word into a table location
 - If that location is already occupied, find a nearby unoccupied one
 - Store p , and (if necessary) enough additional information to reconstruct the word

- Advantages (relative to suffix arrays):
 - only $O(N)$ to construct table, $O(1)$ to lookup an entry
- Disadvantages:
 - less memory efficient
 - requires choice of a fixed word length w
 - (slightly) harder to program