Lecture 2

- Algorithms & computation time
- Finding exact sequence matches 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
- a gigabase ($N = 10^9$) genome can be analyzed even with 1000s of operations per base
  - 1000×10^9 cycles < 20 min

But not *that* fast!

- $O(N^2)$ analysis of a gigabase genome is impractical (***unless*** the constant factor is ***much*** less than 1):
  - 10^{18} cycles $\approx 10^9$ seconds $\approx$ 32 yrs

Important practical considerations with genome-scale data sets

- Compared to CPU operations,
  - *'cache misses'* (non-cache memory accesses) are very slow (100s of cycles)
  - *disk accesses* are even slower (1000s of cycles)
- But both acquire multiple bytes at once; so accessing data sequentially (in chunks) is better than non-sequentially
  - Burrows-Wheeler is slow!
- Using an *interpreted language* may multiply your cycles by a factor of 10 or more

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!
  - and make sure your program doesn’t (unintentionally) do this!

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).

- 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:
  - 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

Algorithmic Complexity

- Basic questions about an algorithm:
  - how long does it take to run?
  - how much space (RAM or disk space) does it require?
- Would like precise function $f(N)$, e.g.

  $$f(N) = .05\ N^3 + 50.7\ N^2 + 6.03\ N$$

  for
  - running time in secs, or
  - space in kbytes,

  as function of the size N of input data set.
- But
  - tedious to derive, &
  - depends on hardware & software implementation details.

- Instead, more customary to give "the" *asymptotic complexity*, i.e. expression $g(N)$ such that

$$C_1 g(N) < f(N) < C_2 g(N)$$

for some constants C_1 and C_2, and N large enough.

- This is written $O(g(N))$, where notation $O()$ means "up to an unspecified multiplicative constant".
  - e.g. for the $f(N)$ above, the dominating term for large N is $.05\ N^3$, so
    - can take $g(N) = N^3$
    - asymptotic complexity = $O(N^3)$.

- Useful as rough guide to performance, ***but*** can be misleading:
  - for small N a different term may dominate
    - e.g. 2^d term in above example more important for $N < 1000$
  - size of constant may be quite important
    - (big difference between .05 and 5,000,000!)
    - e.g. BLAST and Smith-Waterman both $O(N^2)$, but size of constant enormously different
    - 'cache misses' and disk accesses often dominate running time, yet are invisible to complexity analysis (because affect constant factor only)

- Another limitation: time or space requirement may depend on specific characteristics of input data.
- Usually give "worst case" complexity
  - applies to the worst data set of a given size,

  but

  - in biological situations the *average biologically occurring case* is
    - more relevant
    - often much easier than worst case (which may never arise in practice), or even "average case" in some idealized sense.

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 = # DNA ‘words’ of length n
- $\log_4(10^9) \approx 15$

(Average Case) Complexity Analysis of Suffix Array algorithm

- 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 $O(N\log(N))$ steps – doesn't affect the overall complexity).
  - N.B. cache misses are a significant factor!
- 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

Hashtables

- Like suffix arrays, they 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

Nucleotides to numbers

- Let (for example)

  A = 0, C = 1, G = 2, T (or U) = 3

- Then any (short) sequence has corresponding #, e.g:

  AGGC = 0 x 4^3 + 2 x 4^2 + 2 x 4^1 + 1 = 0 + 32 + 8 + 1 = 41

- allows more efficient sequence storage
  - 1 byte per 4 nucs
  - Can be important for some tasks (e.g. assembly of large # reads)
- Can be used for table locations for short word lengths
  - Not ideal: many ‘collisions’

Hashtables vs suffix arrays

- Advantages of hashtables:
  - 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