Today’s Lecture

• (Finding exact matches in sequences using suffix arrays)

• Algorithm generalities / complexity

• Directed graphs, DAGs
Finding perfectly matching subsequences of a sequence

• Idea (*much* more efficient than ‘brute force’ approach):
  – *suffix array* (Manber & Myers, 1990)
  – make list of pointers to all positions in sequence
  – lexicographically sort list of strings that are pointed to
  – process the list: adjacent entries are “maximally agreeing”
Suffix array step 1:
List of Pointers to Suffixes

ACCTGCACTAAACCCTACACTGGGTTCAGAGATTTCCC

p₁  ACCTGCACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₂  CCTGCACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₃  CTGCACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₄  TGCACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₅  GCACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₆  CACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₇  ACTAAACCCTACACTGGGTTCAGAGATTTCCC
p₈  CTAAACCCTACACTGGGTTCAGAGATTTCCC
p₉  TAAACCCTACACTGGGTTCAGAGATTTCCC
p₁₀ AAACCCTACACTGGGTTCAGAGATTTCCC
p₁₁ AACCTACACTGGGTTCAGAGATTTCCC
p₁₂ ACCCTACACTGGGTTCAGAGATTTCCC

::
Suffix array step 2:
View as Strings to be Compared

ACCTGCACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC

\[ p_1 \] ACCTGCACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_2 \] CCTGCACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_3 \] CTGCACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_4 \] TGCACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_5 \] GCACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_6 \] CACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_7 \] ACTAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_8 \] CTAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_9 \] TAAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_{10} \] AAACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_{11} \] AACCGTACACTGGGTTCAAGAGAGATTTCCC
\[ p_{12} \] ACCGTACACTGGGTTCAAGAGAGATTTCCC

\vdots \]
Suffix array step 3: Sort the Pointers Lexicographically

ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC

p_{10}  AAACCGTACACTGGGTTCAAGAGATTTCCC
p_{11}  AAACCGTACACTGGGTTCAAGAGATTTCCC
p_{28}  AAGAGATTTCGCC
p_{17}  ACACTGGGTTCAAGAGATTTCCC
p_{12}  ACGTACACTGGGTTCAAGAGATTTCCC
p_{1}   ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_{7}   ACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_{19}  ACTGGGTTCAAGAGATTTCCC
p_{29}  AGAGATTTCGCC
p_{31}  AGATTTCGCC
p_{33}  ATTTCC
p_{27}  CAAGAGATTTCGCC

\vdots
\vdots
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
• 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
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
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.
  $$f(N) = 0.05 N^3 + 50.7 N^2 + 6.03 N$$

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 (often uninteresting – though important!) hardware & software implementation details.
Algorithmic Complexity (cont’d)

• 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 = \( O(N^3) \).
Algorithmic Complexity (cont’d)

• Can be misleading, since
  – for small $N$ a different term may dominate
    • (e.g. $2^d$ term in above example much 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

• *but* very useful as rough guide to performance.
Algorithmic Complexity (cont’d)

• Cache misses (non-cache memory accesses) and disk accesses often dominate running time, yet are ‘invisible’ to complexity analysis (because affect constant factor only)
Algorithmic Complexity (cont’d)

• Another limitation to complexity analysis:
  – 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.
Algorithmic Complexity (cont’d)

• Proof that a problem is $NP$-hard
  – (has complexity very likely greater than any polynomial function of $N$ and therefore effectively unsolvable for large $N$)
  
  can be useful in guiding search for more efficient algorithms

  but can also be misleading, since
  – we need some solution anyway, for data sets occurring in practice
  – average biologically relevant case may be quite manageable
Directed Graphs

• A **directed graph** is a pair \((V, E)\) where
  – \(V\) is a finite set of **vertices**, or **nodes**.
  – \(E\) is a set of ordered pairs (called **edges**) of vertices in \(V\).

• An edge \((v_i, v_j)\) is said to **leave** \(v_i\) and to **enter** \(v_j\).
  – \((v_i\) and \(v_j\) are vertices)

• **in-degree** of a vertex = \# edges entering it;

• **out-degree** = \# edges leaving it.
Example:

- $V = \{1, 2, 3, 4, 5, 6\}$,
- $E = \{(1,2), (1,3), (2,4), (4,1), (5,3), (3,1)\}$
- Vertex 3 has in-degree 2 and out-degree 1.
Paths and Cycles

- A **path** of **length k** in $G$ from $u$ to $u'$ (vertices) is
  - a sequence $P$ of vertices $(v_0, v_1, \ldots, v_k)$ such that
    - $v_0 = u$,
    - $v_k = u'$, and
    - $(v_{i-1}, v_i)$ is an edge for $i = 1, 2, \ldots, k$.
- A path can have length 0.
- We write $|P| = k$.
- A **cycle** is a path of length $\geq 1$ from a vertex to itself.
- In example at right,
  - $(1,2,4)$ is a path,
  - $(1,3,5)$ is not, and
  - $(1,2,4,1)$ and $(1,3,1)$ are cycles.
Paths and Cycles (cont’d)

• Can join
  – any path $(u, \ldots, v)$ from $u$ to $v$, to
  – any path $(v, \ldots, w)$ from $v$ to $w$

to get a path $(u, \ldots, v, \ldots, w)$ from $u$ to $w$.  

DAGs

• A *directed acyclic graph* (DAG) is a directed graph with no cycles.

• In a DAG, for distinct nodes $v_i$ and $v_j$, we say
  – $v_i$ is a *parent* of $v_j$, and $v_j$ is a *child* of $v_i$, if
    • there is an edge $(v_i, v_j)$
  – $v_i$ is an *ancestor* of $v_j$, and $v_j$ is a *descendant* of $v_i$, if
    • there is a path from $v_i$ to $v_j$

• In a DAG the length of a path cannot exceed $|V| - 1$, because
  – (where $|V|$ = total # vertices in graph)
    – in a path of length $\geq |V|$, 
      • at least one vertex $v$ would have to appear twice in the path;
    – but then there would be a path from $v$ to $v$, i.e. a cycle.
Structure of DAGs

• Define the *depth* of a node $v$ in $V$ as:
  – the length of the longest path ending at $v$;
by above, the depth is well-defined and $\leq |V| - 1$.

• *Every descendant $w$ of a node $v$ has higher depth than $v$:*
  If
  – $(u, \ldots, v)$ is path of length $n = \text{depth}(v)$ ending at $v$, and
  – $(v, \ldots, w)$ is path from $v$ to $w$,
then $(u, \ldots, v, \ldots, w)$ is a path of length $> n$ ending at $w$, so $\text{depth}(w) > n$. 


Structure of DAGs (cont’d)

• Every node $v$ of positive depth has a parent of depth exactly one less:
  – Let $(u, \ldots, v', v)$ be path of length $n = \text{depth}(v)$ ending at $v$.
  – Then $v'$ is a parent of $v$.
  – Since $(u, \ldots, v')$ has length $n - 1$, $\text{depth}(v') \geq n - 1$.
  – Since also $\text{depth}(v') < n$ (because $v$ is a descendant of $v'$), $\text{depth}(v')$ is exactly $n - 1$.

• The nodes on any path are of increasing depth.