**Algorithms for DNA Sequencing**

*Week 1:*

* Two problems: **read alignment problem** and **assembly problem**
* **De novo shotgun problem** – featured in the human genome project; thought it couldn’t be solved in practiced vs. thought could be solved with a fast computer (second was correct)
* **1970s**: Sanger sequencing a.k.a. chain termination method
* **Human Genome project** used this
* But since then (after 2001) cost of sequence has dropped dramatically (logarithmic)
* **2007** – new kind of sequencing technology (next generation or second generation or massively parallel sequencing) – can sequence many millions or billions in parallel
* Cost wasn’t the only improvement – speed, accuracy, ease of use
* How does a DNA sequencer sequence a genome? – Bad at reading LONG sequences, good at reading lots of SHORT sequences
  + Reads lots of middle bits of the DNA (snippets called sequencing reads or reads)
  + Usually reads are 150 bps, but lots and lots and lots – cover the whole genome over many times
* Sigma is the set of finite characters in our string, epsilon represents a string of length zero
* **Suffix**: a string that ends at the end of the longer string, there are n suffixes in a string of length n
* **Offset n**: the item at position n
* **Second gen sequencers** ‘watch’ what DNA is doing as it replicates

**How Second Gen Sequencers Work**:

* Blood sample: extract DNA from cells, chop into single-stranded snippets
* Deposit them onto a slide so that they’re standing up – molecules to sequence, one read per template
* Add DNA polymerase and bases with terminators, it can add only one base
* Take a photo of this – they glow a particular color and tell which base was added to each template
* Next, cleave the terminators and then repeat
* At the end, there is a series of photos – one per sequencing cycle
* If you look at just one template, you can read off the series of colors observed so then we can take the compliment of those bases and get sequence of template strand
* Important: billions on slide, can photo graph at all at once (massively parallel, and terminators keep in sync

**Sequencing errors and base qualities**:

* Before we add bases or polymerases, make a cluster of the same template around each original template
  + Clusters because when we take the photograph, there wouldn’t be enough from a single template
  + PROBLEM: what if there is a non-terminated base? It won’t block the polymerase and you get more added ahead of schedule (out of sync)
  + Some are on schedule, some that are ahead of schedule – as we move on, number that are out of sync grows
  + Software: analyzes images (base caller) and has to deal with ambiguity
  + If it isn’t confident, we can know which we aren’t so sure about
  + Base caller reports **base quality** – estimate of probability that the base was called incorrectly
  + Confidence tends to decrease towards the end
  + Q = -10 \* log10p (Q is the base quality and p is the probability that the base call is incorrect)
  + Why not report p? The scale Q is on makes it easier to interpret (if Q = 10, 1 in 10 its incorrect, 20 => 1 in 100, 30 => 1 in 1,000)
  + How to get Q? p = non-orange light/total light (if orange is most prevalent, what fraction contradicts)
  + In reality, this is more complicated

**Sequencing reads in FASTQ format**:

* Four lines of information => name of read (information about it), sequence of bases as reported by base caller, placeholder, sequence of base qualities
* A FASTQ file is a series of these reads
* **Base Quality Line**: match up with corresponding characters in sequence line, each base quality is an ASCII encoded version of Q (there is a table)
* **Method for BQ to ASCII**: round Q to nearest integer, add 33, then turn into corresponding character according to table – called **Phred 33**

**Analyzing Sequencing Data**:

* Can’t answer just by looking at the reads
* First problem: take reads and glue them back together somehow – like putting together a puzzle (you have pieces AND a picture of the puzzle)
* Unrelated genomes are 99.8-99.9% similar in humans – only 1-2 difference per 1000 bases so you can use them as a template/guide (Human Reference Genome)
* Take a sequencing read and hold it up to the reference sequence to find where it matches
* First half of the course: solving this **read alignment problem** – given the a read and a reference, where does it align?
* Second half: without reference genome – try to find where the overlap, **assembly problem**

**Read alignment challenges**:

* Repeatedly taking a sequence read and a reference genome and looking for the place where it matches most closely, do for each read (billions)
* Human genome is 3 billion long
* Interpret these as strings – there is a lot of stuff for working with strings

**Exact matching problem:**

* At what offsets does **patter P** occur within **text T**
* This is a simplified version of the read-alignment problem
* In python, use find to find leftmost occurrence

*Questions:*

Q: If x = length of P and y = length of T, how many alignments are possible given x and y?

A: y – x + 1

Q: What’s the greatest number of character comparisons possible?

A: x\*(y – x + 1) (WORST CASE ANALYSIS)

Q: When would above worst case happen?

A: If it just kept matching – every character of P matches every character of T (RARE)

Q: What is the least # possible?

A: y – x + 1 (for each, you only have to do one)

Q: When?

A: First character is nowhere in longer text

Q: How many do we do in this example?

A: length of the string + 2 (word and would) + 4 (for the actual word) = 46

NOTE: closer to minimum (41) than maximum (164)

*Week 2:*

Read alignment problem

* Previously: naïve – not fast
* Discuss Boyer-Moore, indexing, pigeonhole principle

**Boyer – Moore**

* Similar to naïve exact matching but skips alignments it doesn’t need to examine
* Benchmark for exact matching algorithms
  + Example: u doesn’t occur in P, so can skip next two alignments
* **Principle**: learn from character comparisons we do to skip non-fruitful alignments
  + Try alignments in left to right order but try character comparisons in right to left order
* **Bad character rule**: upon mismatch, skip alignments until:
  + (a) mismatch becomes a match or
  + (b) P moves past mismatched character
* **Good suffix rule**: let t = substring matched by our inner loop, skip until
  + (a) there are no mismatches between P and t or
  + (b) P moves past t
  + NOTE: suffix must be at least one
* Putting together
  + With a mismatch, try both rules and take max shift of the two
* How much better is B-M? In this example, we ignored 11 characters and skipped 15 alignments.
  + Generally substantially faster
* In order to use B-M, we need to be able to look up how fast we were able to skip
  + Build lookup tables beforehand
  + To build tables, need pattern P but not T

**Repetitive Elements**

* Real genomes are very unlike “random” genomes
  + Very repetitive
  + For instance, **transposable elements** – 45% of human genome is covered by transposable elements
  + **Alu**: occur more than a million times in the genome (11%)
* Create ambiguity – what if we have an Alu in our alignment?
* If we have non – repetitive bits, its easier b/c we have less to search

**Preprocessing**

* Naïve algorithm doesn’t use pre-processing
* Boyer-Moore uses p to make lookup tables for BC and GS rules
* Costly to do once, but makes up over time
* Preprocessing text T? Sounds like a lot of work because it’s very long…
  + If you’re going to use it again and again, might be worthwhile
* An algorithm that preprocesses T is **offline**, otherwise algorithm is **online**
* Scenario: Naïve Algorithm is online
* Scenario: Boyer-Moore is online
* Scenario: web search engine is offline b/c text T is enormous (entire web)
* Scenario: read alignment is offline b/c we have many snippets we’re trying to align to a single human DNA sequence (P changes but T doesn’t)

**Indexing and the k-mer index**

* Read alignment requires an offline algorithm!
* Example: for a book, preprocessing involves creating an index (alphabetized key terms with associated page #’s)
  + **Querying index**
* Example: grocery store items **grouped** into aisles
* Ways to preprocess: ordering or grouping to make querying easier
* Indexing DNA is more like ordering in a book
  + Take each substring of a certain length and associate the starting offset
  + First letter is alphabetized
  + If a substring is the same, append to list of offsets
  + **K-mer**: substring of length k (in ex 5-mer index)
* How to query?
  + With P, take first k-mer and query the index
  + Index replies where its seen it
  + Does the rest of P match? => do verification with final characters
* Variation: take the second 5-mer (doesn’t really make a difference!)
* **Index hit**: for each offset the index returns back – not always a match

**Ordered structures for indexing**

* Discussed data structure is a **multi-map** (multi because multiple offsets)
* Two data structures we can use
* Type 1: based on ordering: key, value pair
  + Get offsets, order alphabetically
  + Query using binary search (log(n))
  + Python has binary search functions (import bisect)
    - bisect.bisect\_left(a, x): leftmost offset where x can be inserted into a to maintain sorted order

**Hash tables for indexing**

* Other method for implementing multi-map
* Hash table starts out with an array of empty **buckets**
  + As you add items, boxes become lists
  + **Hash function** maps each distinct key (3-mer) onto one of the buckets (like aisles)
  + h(3-mer) = bucket (a linked-list)
  + When adding another to same bucket, add it to the end of the linked list
  + Due to pigeonhole, you end up with multiple things in each bucket (**collision**)
* **Querying**
  + Run hash function to map to bucket (only bucket to look in)
  + Check linked list
  + Python dictionary is a hash table!

**Variations on k-mer indexes**:

* What if you didn’t use every k-mer? Every other k-mer?
  + Index is now smaller and a little faster to query (save on bisection)
  + Is this concerning? Will you miss matches?
  + No – just use the first and second k-mer (or any even/odd pair)
  + Generalization: every nth kmer, query n offsets (0modn…n-1modn)
* Build over subsequences rather than substring
  + Subsequences are nonconsecutive but in correct order
  + Shape of subsequence remains the same
  + Extract same shaped subsequence from P then query that way
  + Advantages: tends to increase specificity of filter

**Genome indexes used in research**:

* Building fast, smaller, more flexible indexes in an active area of research
* Subsequence filter- larger fraction of index hits leads to matches
* **Suffix index**: take every suffix from text T and organize into a data structure
  + Idea: alphabetize and use binary search
  + Sadly that’s fucking huge
  + Represent a subset into the genome as an offset into the genome – **suffix array**, must also store T
  + Storing two things of length n
* There are other types of suffix indexes: **suffix tree** (uses grouping) and **FM Index** (based on **Burrow-Wheelers transform**) – extra compact
* If used each to build index for human genome, how much size?
  + Suffix tree > 45 GB, suffix array > 12 GB, FM Index ~ 1 GB
* **Bowtie** uses FM Index, for instance

**Approximate matching, Hamming and edit distance**:

* Previously, all for exact matching but there are often differences b/w read and reference due to
  + Sequencing error
  + Natural variation
* Need algorithms for **approximate matching**: mismatches/substitution, insertions, deletions
* Want to be able to talk about the **distance** between two strings –how different they are
* **Hamming distance**: minimum # substitutions needed to turn one into the other
* **Edit distance**: minimum # edits (substitutions, insertions, deletions) to turn one into the other
  + X and Y don’t have to be same length unlike with Hamming
* If we change naïve matching and specify Hamming distance
* Can we adapt Boyer-Moore?

**Pigeonhole Principle**:

* Want: way to apply exact matching to approximate matching
* If you partition p into u and v, if P occurs in T with 1 edit, then u or v appears with no edits
* If P occurs in T with up to k edits, at least one of the k+1 partitions must appear with 0 edits
* Due to pigeonhole principle! If you have k edits, and k+1 holes…at least one hole doesn’t have an edit
* Process: divide p into k partitions, run exact matching on T, then run verification step
* Note: we can use all the exact matching algorithms to find approximate matches in this way

*Week 3:*

**This Week**:

* Dynamic programming for edit distance
* Move on to genome assembly

**Solving the Edit Distance Problem**

* Previously: Pigeonhole principle for approximate matching
* Now: **dynamic programming**/**edit distance**
  + **Global** v **Local**
  + Don’t depend on exact matching algorithms
* Hamming Distance and Edit Distance
  + Hamming is substitutions - easier
  + Edit is substitutions, insertion, deletion
* How to calculate edit distance??
* If |X| = |Y| what can we say about relationship between editDistance(X,Y) and hammingDistance(X,Y)?
  + editDistance(X,Y) <= hammingDistance(X,Y)
* If |X| != |Y| what can we say about the edit distance(X,Y)?
  + editDistance(X,Y) >= ||X|-|Y|| - must at least be enough to make same length
* **Algorithm for finding edit distance**
  + Take two strings, want to know edit distance between them
  + Find edit distance between **prefixes** of them – then calculate entire strings
  + Call prefixes a and b
    - edist(aC, bA) = min (edist(a,b) + 1, edist(aC, b) + 1, edist(a, bA) + 1
      * just add A,C onto end, aC into B then add A, edit alpha into bA and then insert C
    - edist(αx, βy) = min{edist(α, β) + δ(x,y), edist(αx, β) + 1, edist(α, βy) + 1}
      * δ(x,y) = 0 if x = y, 1 o/w
    - ^ **Recursive function** – really really slow

**Dynamic Programming for Edit Distance**

* Generate handy dandy matrix
  + X as rows
  + Y as columns
  + First row/column are the empty string
* Algorithm fills in matrix with edit distance between corresponding prefixes
* Lower right corner is the complete edit distance
* Fill in using the edist expression from above
  + Start with SHORT prefixes and move through
  + Case 1: up to left (diagonal)
  + Case 2: to left
  + Case 3: up
* Need to also fill in first row/first column
  + Edit distance between empty string and string
* HOW FAST: quite fast!
  + Doesn’t re-call for exact same arguments – once and only once

**A new solution to approximate matching**

* P: label rows
* T: label columns
* Difference: P by T + 1
  + Initialize first row as all 0’s
  + First column initialized the same way
* Fill in the same way
* Look in final row: P occurs in T with 2 edits
  + Closest match between P and some substring of T
* To find location: ask ‘how did we get here?’
  + **Tracing Back**
  + Tells you which substring of T matches with P
  + Also tells you the shape of the alignment – where are the differences between P and T
* Problem: proportional time to size of matrix
  + Some variations of Boyer Moore are linearly proportional
  + But Boyer Moore can only do exact matching
  + This may be efficient for use on its own
  + Typically used in combination with other techniques – pigeonhole, indexing…
  + Filter then use this

**Global and Local Alignment**

* Variations on theme for DP for edit distance
  + Previously, edit distance penalized all things equally
  + Might not always be useful – some DNA substitutions are more likely than other for instance
    - Transition (A to G or C to T) are two times more likely than transversions in terms of frequency
    - Small gaps in the genome are rarer than substitutions
* Penalty Matrix: element for every kind of penalty
  + Substitutions – transitions/transversions
  + Gaps
  + Change for value added on ED formula – call penalty function
* **Global Alignment**: what we have done
* **Local Alignment**: not trying to find distance between two strings OR one string to a substring of another
  + Rather, find most similar pair of SUBSTRINGS from X and Y
  + Must consider all possible substrings of X and Y – is this a hellish problem?
  + No different than global alignment
  + Instead of penalty matrix, use scoring matrix - give bonus for match and penalty to all other things
  + **To find local alignment from matrix**, look for maximal element in matrix
    - Use usual trace back but STOP at 0

**Read Alignment in the field**

* Indexing and DP
* **Indexing** lets you hone in on candidate locations in genome that share a substring with P
  + Indexing is a filter
  + If you only use DP, it takes a seriously long time (years) and matrix is huge
* Indexes are bad a mismatches and gaps so then you can use another algorithm for verification step
  + Ideal for dynamic programming – flexible and can handle mismatches and gaps

**Assembly working from scratch**

* This is the case where you don’t have a reference genome
* **De novo/shotgun assembly**
* Fundamentally hard problem
  + More computationally challenging than read alignment

**First and Second Laws of Assembly**

* **Assembly Problem**: given many sequence reads, reconstruct the genome
* **Coverage**: amount of redundant information we have in our genome
  + This position has a coverage of 5 – 5 pieces of evidence for that base
  + They may not always agree!
* **Overall Coverage**: coverage averaged over all genome
  + Total length or reads/length of the genome
  + EX: 177/35 is about 7-fold coverage
* **First law of assembly**: if a suffix of read A is similar to a prefix of read B then A and B are likely to overlap in the genome
  + Why might there be mismatch in these overlaps?
    - Sequencing errors
    - Polyploidy – copies of chromosome, may differ
* **Second law of assembly**: more coverage leads to more and longer overlaps
  + Important because overlaps are the ‘glue’

**Overlap Graphs**

* Way of representing overlaps in one big structure – a **directed graph**
  + Nodes/Edge (directional)
* Can represent the overlaps like:
  + Each node is a read
  + Draw edge A -> B when suffix of A overlaps prefix of B
  + EX:
    - Nodes: all 6-mers from genome
    - Edges: overlaps of length >= 4 (**should define size**)
    - Label edges with the length of the overlap
* To find overall sequence, there is a way of walking along the graph
  + Walking along longest edges?

*Week 3:*

**Shortest common superstring problem**:

* Solves genome assembly problem imperfectly
* **SCS**: given set of strings S, find SCS(S) – shortest string containing the strings in S as substrings
* Suppose we have an algorithm for this:
  + Solution to SCS(S) is an assembly of the genome
* Problem: there are no efficient algorithms for solving this
  + NP – complete
* **Algorithm**:
  + Idea: pick order for strings in S and construct superstring
  + Find longest overlaps and glue together
    - EX: AAA, AAB -> AAAB
    - AAB, ABA -> AAA
    - And so on…
  + After this, we have a superstring
  + Different orders result in different superstrings
  + To find shortest: try all orderings, pick shortest
  + PROBLEM: n! orderings possible

**Greedy shortest common superstring**:

* Might not result in an optimal solution
* Visualize as graph as described before
* Proceed in rounds, pick edge corresponding to longest overlap and merge nodes on either side of that edge
  + Greedy: always picking longest overlap
* **Algorithm**:
  + Pick edge with longest overlap and merge nodes (the overlap)
  + Again, pick edge w/ max overlap and merge again
  + Continue…
  + If there are leftover nodes, concat them together

**Third Law of Assembly: repeats make assembly difficult**

* When the genome is repetitive, SCS might not be the genome
  + Result: collapses the repeats down – can’t tell how many copies of the repeats there should be
  + Ambiguity
* HUGE IMPORTANT ISSUE
* Other algorithms might make different mistakes due to repetitive sequences
* Half the genome has repetitive DNA sequences
* De Bruijn might help, new technologies might help

**De Bruijn Graphs and Eulerian Walks**:

* **De Bruijn Graph**:
  + Directed, multi graph
  + Assume reads are all k-mers of genome, exactly once
    - Iffy assumption
  + Examine each k-mer and make some additions to graph:
    - Extract left and right k-1-mers
    - Add node for left, add node for right (unless already there)
    - Make edge from left to right
  + Notes:
    - 1 edge per k-mer
    - 1 node per distinct k-1-mer
  + You can reconstruct the original genome from the graph – a walk through the graph
    - Cross each edge exactly once
* **Eulerian Walk**: a walk crossing each edge exactly once
  + Not every graph has one, but one that does is a **Eulerian Graph**
* **Algorithm**:
  + Take reads, built graph
  + Eulerian walk gives genome!

**When Eulerian walks go wrong**:

* In one example, didn’t over-collapse a repeat!
* But cannot escape the third law…
* What if you can go multiple ways? Multiple walks?
  + Uh oh…
  + There is ambiguity!
  + Generally for repetitive genomes, there are many walks – others are incorrect reshufflings
* Decreasing k-mer length – affected by repeats
* Other issue: the one read per k-mer assumption (w/o errors)
  + These don’t happen
  + Different lengths, uneven coverage…
  + Can still use procedure, but split each read into k-mers
  + BUT the graph isn’t Eulerian (necessarily or in practice)
* DB graphs are a common way of representing reads and the assembly problem

**Assemblers in Practice**:

* Overlap graphs and de Bruijn graphs
  + Still use these graphs
  + Two different categories: overlap-layout-consensus assembly and De Bruijn graph assembly
  + Start by building graph
* Graph typically ends up being big and messy and walk isn’t obvious
* Why messy?
  + Sequencing errors: dead ends and disconnected bits
  + Can contain edges that tell us nothing (some edges can be inferred from another)
  + Polyploidy – maternal and paternal DNA for ex
    - Don’t want to get rid of, keep a note
  + Repeats again – create ambiguity
* How to deal?
  + Chopping assembly into pieces
  + Look at pieces of graph where there is no ambiguity
    - **Contigs**: portions that can be reconstructed unambiguously
  + Assembly reports contigs
  + Human genome still has some holes in it

**The future is long**:

* There are ways (in theory) to combat
* MAKE THE READS LONGER: anchor the repetitive sequence with non-repetitive context
  + Bigger pieces in a puzzle
  + Span all copies of a repeat in a k-mer
* How long does it have to be?
  + Long enough to span the repeat and some sequence on either side
* How do we GET longer reads?
  + Hard, technologically speaking
  + Innacuracy
* **Paired end sequencing**:
  + Similar technology to NextGen
  + Instead of sequencing one end of the template, sequence both ends
  + Could have part of the middle as a gap
  + BUT can use similar methods as described even with missing sequencing in middle
  + Twice as many bases!
* There are some methods that can improve this, but its slow
  + Tiny camera to eavesdrop on DNA polymerase
  + Nanopores and electrical current
  + Sequence one molecule at a time unlike nextgen
  + PROBLEM: these are pretty error prone (10-15%)