A set of notes by Charlie **Spring 2017**

Notes on 'Algorithms for DNA Sequencing'

Fourth Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week One: DNA sequencing, strings and matching

Most importantly, we'll learn how genomes can be represented as strings and substrings. There are two main technical problems: a.) read alignment, and b.) assembly. This is such a timely area of study because sequencing has become so cheap. DNA encodes your genome: the sum total of all your genes. Your genome is a recipe book - for the machines which do the work of building and maintaining you. The recipes are written in As, Cs, Gs and Ts. A-T and C-G (these are 'bases'). The fact that we can write DNA molecules as a string has huge implications. DNA sequencers are good at reading lots of short stretches of DNA to produce 'reads' which are many magnitudes of order shorter than DNA.

A string `S` is a finite sequence of characters - a set of Σ={A,C,G,T} in this case. $|S|$= the number of characters in `S` - `len(S)`. ε is the empty string: `len('')`. The left-most offset is 0. The concatenation of two strings glues them together: e.g. s + t. A substring of S is a string occurring inside S. e.g. s[0:6] is the same as s[:6] - if the zero is ommitted, it is an implied zero. A suffix occurs at the end of S. In Python: double and single quotes represent the same thing. Index with square brackets. `len('')` tells the length, and `print()` prints them. `.join()` joins a list together into one string. `random.choice('ACGT')` will randomly pick bases. Looping through with an underscore can be used when we dont need to reference the looped variable in the loop e.g. `for _ in range(0,10)`. Example function to find longest common prefix between two strings:

```
def longestCommonPrefix(s1,s2):
    i=0
    while i < len(s1) and i < len(s2) and s1[i]==s2[i]:
        return s1[:i]
```

A double equal sign tests equivalence. How about an example to get a reverse compliment? Create a dictionary called compliment to get a reverse: `compdict=['A':'T', 'C':'G','T':'A','G':'C']`. It is 'FASTA' files which contain these base sequences. **Cool trick - ! allows us to use command line within ipy notebooks!** For example: `!wget .....` How to read genomes into strings, and then print it out the first hundred bases with `genome[0:100]`.

```
def readGenome(filename):
    genome =''
    with open(filename,'r') as f:
        for line in f:
            if not line[0] == '>':
                genome+=line.rstrip()
```

How about we count the bases? `import collections`, `collections.Counter(genome)`. How do second generation sequencers work? Double stranded DNA gets split in half to act as a template. DNA polymerase takes bases to build the complimentary strand (of the template strand) piece by piece. Make DNA single stranded: deposit them into flat suraces (like a slide). Snap photo of terminated bases, iterating the process, to get a series of photos: one per sequencing cycle. One dot per template strand - billions of single stranded templates on a slide - photographing all at once, terminators keep the strands in sync and give us time to snap the photograph. However, there are sequencing errors potentially because a base is not terminated or is ahead of schedule. For each base call, the base caller reports a score on the probability that the base type is accurate. Reads are encoded within a FASTQ format. The second line is the sequence of bases, and the fourth line is the line of base qualities. Each base quality is an adjusted version of the probability that the base call is incorrect: $Q = -10log_{10}p$. Each value is ASCII encoded: each value maps to an integer on a corresponding lookup table. The most common way of encoding is 'Phred+33': take a value from the ASCII table and add 33 to it. Each FASTQ file comes in sets of 4.

```
def readFastq(filename):
    sequences = []
    qualities = []
    with open(filename) as fh:
        while True:
            fh.readline()
            seq=fh.readline().rstrip()
            fh.readline()
            qual=fh.readline().rstrip()
            if len(seq) == 0 ;
                break
            sequences.append(seq)
            qualities.append(qual)
    return sequences qualities
```

Convert Phred33 to their corresponding quality scores:

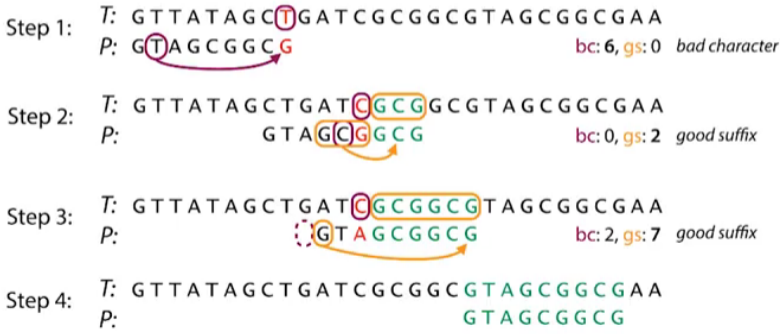
```
def phred33toQuality(qual):
    return ord(qual)-33
```

Why are we interested in the GC content? It's different from species to species - we are trying to figure out if the mix of different bases is moving along the read - can be useful for quality control to see if anything strange is happening. Note that the average human genome GC content is higher than 0.5. The base caller can sometimes report an N - when it has no confidence in the read.

```
def findGCbypos(reads):
    gc=[0]*100
    totals=[0]*100
    for read in reads:
        for i in range(len(read)):
            if read[i]== 'C' or read[i] == 'G':
               gc[i]+=1
               totals[i]+=1
    for i in range(len(gc)):
        gc[i]/=float(totals[i])
```

However, to infer meaningful information from these snippets, we must take these reads and stitch them back together to infer the sequence of the DNA. We do this with a reference genome - this is typically called the 'read-alignment' problem (unless we have no reference - then we use 'de novo' assembly). We're looking for the place where the sequence looks most closely to the reference genome (which is about 3 billion bases long). In Python, `string.find('word')` returns the index of the first occurrence.

Steps to Suffixing



```
def naive(p,t):
    occurrences=[]
    for i in range(len(t) - len(p)+1):
        match = True
        for j in range(len(p)):
            if not t[i+j] == p[j]:
                match = False
                break
        if match:
            occurrences.append(i)
```

Sequencing errors and the fact that we are not using the correct reference genome mean that we might not match a high percentage. We should also try and align the reverse complement of the DNA also (i.e. is 'strand aware'). However, these techniques are restrictive in that they are focusing on exact matches: we want to allow for approximate matches through sequencing errors.

Week Two: Preprocessing, indexing and approximate matching

The naive algorithm is slow and only considers exact. Another choice: Boyer-Moore - fast, practical and simple. Indexing is the reason why you type a query string into a search engine, you get a result back instantly. **Boyer-Moore**: Similar to naive exact matching - it skips many alignments that it doesn't need to examine - the benchmark exact matching algorithm. (i) It learns from character comparisons to skip pointless alignments - try alignments in left to right order, but try character comparisons in right-to-left order. (ii) The bad character rule of Boyer-Moore: upon a mismatch, skip alignments until a mismatch becomes a match, or the substring (P) moves past the mismatched character. (iii) The good suffix rule: skip until there are no mismatches or P moves past t ('tries to keep the good matches a match'). Each of these rules will tell us some amount that we can shift P - take the maximum of the two.

Repetitive elements of the genome creates ambiguity for the read-alignment problem - where does our read come from? For example: about 10% of the human genome is covered by Alu repeats.

```
def boyer_moore(p,p_bm,t):
    i=0
    occurrences=[]
    while i< len(t)-len(p)+1:
        shift=1
        mismatched=False
        for j in range(len(p)-1,-1,-1):
            if not p[j]==t[i+j]:
                skip_bc = p_bm.bad_character_rule(j,t[i+j])
                skip_gs = p_bm.good_suffix_rule(j)
                shift = max(shift, skip_bc, skip_gs)
                mismatched=True
                break
        if not mismatched:
            occurrences.append(i)
            skip_gs=p_bm.match_skip()
            shift=max(shift,skip_gs)
        i+=shift
    return occurrences
```

The BM algorithm pre-processes to build lookup tables to make it faster for bad character and good suffix rules - this 'amortizes' the cost over time. An algorithm which does pre-process the text (T) is called 'offline' (one that does is 'online'). For example, the naive algorithm is 'online', BM is 'online' because it `preprocesses only the pattern P`. Two concepts which are useful: 'ordering' as per the index of a book, and 'grouping' as per the areas of a grocery store. When p matches within T, we call it an occurrence - but not all index hits lead to matches. A 'k-mer' index - an index built by taking all k-mers of text T and adding them to a

data-structure ('multi-map') which relates to them where they occurred in the text - it associates keys with offset values in the genome. We then order the indexed datastructure, and query the index with this 3-mer. The number of queries that we need to make is approximately equal to $\log_2(n)$ bisections. Python makes a bunch of tools for binary search such as the `import bisect` module. For example: `bisect.bisect_left(a,x)`: gives the leftmost offset where `x` can be inserted into `a` in order to maintain order. Hash tables can be used to implement multi-maps: used to represent sets and maps in practice. The hash-function h maps 3-mers (for example) to 'buckets' in the hash table - we then append the corresponding key-value pair to that bucket. The python dictionary type is a type of implementation of a hash table. For example:

```
t='GTGCGTGTGGGGG'
table={'GTG':[0,4,6],'TGC':[1],'GCG':[2], 'CGT':[3],'TGT':[5],'TGG':[7],'GGG':[8,9,10]}
```

Before, we built our map table of every k-mer. What if we didn't take every k-mer? What if we took every other k-mer (such as those which started only at even-offsets?). This makes the index smaller, and is a little faster to query - but wont this cause us to miss some of the matches? What about if we only chose every n-th k-mer? We can also build this over subsequences. Substrings are always subsequences, but subsequences are not always substrings. This kind of technique can increase the *specificity* of the filter provided by the index - when we get an index hit it is going to lead to a correct verification more of the time than if we had taken the characters to be consecutive. Indexes used in genomic research: another idea - suffix index. Instead of extracting every substring and putting it into an index, we could take every suffix - all suffixes in alphabetical order - query it using binary search. This can be represented by just one integer - the *offset* at which it occurs - this leads to a more manage-ably sized data structure (growing linearly, rather than quadratically). Another example is the suffix tree (like the suffix array), but organizes them using the principles of grouping (whereby the suffix array puts everything in order). Another type is the FM index: based on the BW transform: this is much more compressed and can easily be fit into memory on a local computer (commonly used in BowTie and BowTie2). All of these methods are *exact*. But we might want *approximate* methods due to sequencing errors or natural variation. There might be deletions, insertions or substitutions. There are multiple of these measures of 'distance': e.g. for strings X and Y where ($|X| = |Y|$), the **Hamming distance** is the minimum number of substitutions needed to turn one into another. The **Levenshtein** distance is the minimum number of edits (substitutions, insertions and deletions) needed to turn one string into another. We can very easily modify the naive exact algorithm above to turn it into a naive Hamming algorithm as follows:

```
def naiveHamming(p,t,maxDistance):
    occurrences=[]
    for i in xrange(len(t)-len(p)+1):
        nmm=0
        match=True
        for j in xrange(len(p)):
        if t[i+j]!=p[j]:
            nmm+=1
            if nmm>maxDistance:
                break
            if nmm <= maxDistance:
                occurrences.append(i)
    return occurrences
```

The pigeonhole algorithm is a method which allows us to extend our exact algorithms to approximate. For example: in the one edit case - if we split P into u and v - then either u or v will be an exact match. This can be extended to k-edits - **at least** one of these k+1 partitions must appear with no edits. The pigeonhole principle - 10 pigeons, but 9 holes. In our problem, we have 9 holes, but 8 pigeons. It is this bridge which allows us to combine our approx algorithms with the aforementioned techniques (i.e. BM, with respect to hits and occurrences, etc). For example: if we're searching for approximate matches of P within T allowing up to k mismatches, then we should first divide P into k+1 partitions, and at least 1 partition will have no mismatches.

Week Three: Edit Distance, Assembly, Overlaps

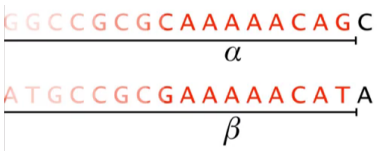
Dynamic programming algorithms allow us to look for approximate occurrences of a pattern in text, including occurrences with insertions and deletions and allow us to calculate flexible string similarities with appropriate penalties. The Hamming distance is actually quite easy to to calculate, as shown in the function above. How about an algorithm to calculate the edit distance between two strings? More complicated. What's the relationship between the Hamming and edit distance? Are they equal? Is one greater or equal? The edit distance will always be less than or equal to the Hamming distance. Can we put a lower bound on the edit distance between X and Y? In order to edit X and Y, we at least have to introduce as many edits to make them the same length - and then more maybe to make them the same sequence. Tactical insertions potentially allow us to reduce the distance to zero much more quickly than substituting and deleting.

This function (delta) is like a recursive function, as shown by the Python code. However, these functions are typically extremely computational and take a very long time - e.g. `edDistRecursive('ABC','BBC')` this initial call will make 3 recursive calls, which will each make 3 more recursive calls, etc. However, a lot of these calls are the same - and it would be useful to remember what calls are made. In order to prevent this, we can rewrite the function in terms of a matrix where the characters label the rows and columns: each column corresponds to a particular prefix (where first row and column is the empty string). This type of dynamic programming is very useful for calculating these types of sequencing applications.

```
def editDistance(x,y):
    D=[]
    for i in range (len(x)*+1):
        D.append([0]*len(y)+1)
    for i in range(len(x)+1):
        D[i][0]=i
    for i in range(len(y)+1):
        D[0][i]=i
    for i in range(1,len(x)+1):
        for j in range(1,len(y)+1):
            distHor=D[i][j-1]+1
            distVer=D[i-1][j]+1
            if x[i-1]==y[j-1]:
                distDiag=D[i-1][j-1]
            else:
                distDiag=D[i-1][j-1]+1
            D[i,j]=min(distHor,distVer,distDiag)
    return D[-1][-1]
```

Edit distance allows us to find the distance between two strings, but what about if now apply it to approximate matches? Revert to looking for patterns (P) in a string of text (T). Now, initialize the first row with all 0s and the first column as ascending integers as before. We can then use the 'traceback' path to find the substring which has the minimum number of potential edits. This traceback path also tells us the shape of the vertical alignments (having looked in th final row to find the smallest number of edits). This is a lot of work: proportional to the size of the matrix (len(P) times len(T)) - with no skipping. This algorithm is not practical on its own for the read alignment problem. It tends to be used in addition to other techniques, such as indexing, the pigeonhole principle, etc. We can use these dynamic programming tools for both global and local alignment problems. Global alignment: edit distance penalizes all types of edits the same amount - no difference between substitution and insertion, etc. What if certain base to base substitutions were more likely, and we wanted to penalize them less? This is the case with genomic sequencing: we can divide all DNA substitutions into two categories: transmissions and transversions. For substitutions that convert a purine to a purine or pyrimidines: this is a transition - all other are transversions. In reality, transitions are twice as frequent as transversions - we might want to penalize transversions more then. With respect to the human reference genome, the substitution rate is something like 1 in 1000, the indel (insertion/deletion) rate is 1 in 3000. To incorporate this, we can use something like a penalty matrix. Incorporating this into our edit distance function is very simple: instead of our delta function or adding 1, for example, just add the relevant lookup from our penalty matrix. Global alignment gives the user the ability to set penalties applicable to the

Edit Distance Suffixes



α x

β y

```
def edDistRecursive(a, b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

$$edist(\alpha x, \beta y) = \min \begin{cases} edist(\alpha, \beta) + \delta(x, y) \\ edist(\alpha x, \beta) + 1 \\ edist(\alpha, \beta y) + 1 \end{cases}$$

$\delta(x, y) = 0$ if $x = y$, or 1 otherwise

biological problem at hand. Local alignment - we're trying to identify the substring of x and y which are most similar to each other. Instead of using a penalty matrix - use a scoring matrix, where we give a positive bonus for a match and a negative penalty for all other types of differences. We can then use the same traceback procedure as above where we stop when we reach an element whose value is zero. (Summary: Local alignment is compared with similiarities between substrings of x and y). To modify the `editDistance` function above to a global alignment one, all we need to do is define a score matrix and add the appropriate elements of this matrix to each distance when a character is skipped, substituted, etc.

What do the read alignment tools do in practice? They make use of indexing and dynamic programming in conjunction. Indexing allows us to rapidly hone in on candidate locations and acts like a filter. However, in practice, our matrices for each read are huge, because the reference genome is huge also and this would take years without an index. Indexes are really good at finding exact matches, and then use dynamic programming to figure out whether the pattern as a whole has an approximate match to that index hit. On the one hand, the index is very fast and good at narrowing down the space of places to look, but it doesn't handle mismatches and gaps naturally at all. And on the other hand, dynamic programming does very naturally handle mismatches and gaps (but is slow).

Until now, we've been focusing on the problem of 'alignment' - where we're given a reference genome. The assembly problem relates to when we have no reference - this is also called 'de novo'/shotgun assembly. The figure shows us the task behind the goal of assembly. One important concept: coverage - which relates to the number of reads we have over one base in the genome - i.e. the number of 'votes' for what a specific base should be. We can average this into the 'overall coverage'. How can we piece together the genome sequence? **The first law of assembly: If a suffix of read A is similar to a prefix of read B, then A and B might overlap on the genome**. Why are there possibly differences in coverage over one specific base? i.) sequencing errors, ii.) polyploidy - humans have 2 copies of each chromosome - and copies can differ because they have different bases at that position. **The second law of assembly: more coverage leads to more and longer overlaps.**. The way to approach this problem is through one big structure - because the overlaps involve the same read - we need to build a directed graph where the nodes and edges have meaning. But what do we mean by an overlap? Some overlaps are less convincing than others: a length 1 overlap could be just a coincidence which we don't want to consider an overlap - we need to set a threshold and say as long as the overlap is more convincing, we can count it as an overlap and draw the corresponding directed edge (e.g. have an overlap of at least length 4). We can walk through the nodes to infer the sequence of the original genome. An example function is something like:

```
def overlap(a,b,minlength=3)
    start = 0
    while True:
        start=a.find(b[:minlength], start)
        if start == 1:
            return 0
        if b.startswith(a[start:]):
            return len(a)-start
        start+=1
```

```
def naive_overlap_map(reads,k):
    olaps{}
    for a,b in permutations(reads,2):
        olen=overlap(ab,minlength=k)
        if olen>0:
            olaps[(a,b,)]=olen
    return olaps
```

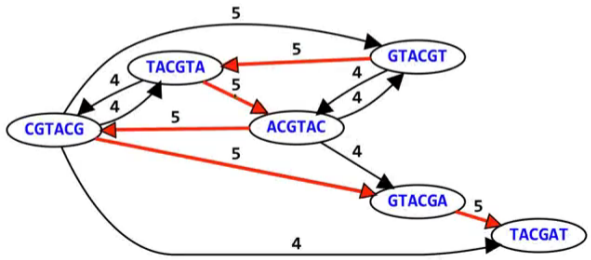
Week Four: Algorithms for Assembly

The problem that is critical in assembly is the 'shortest common superstring' (SCS) problem - the shortest possible string that contains all of our input strings as substring. For example, given a set of strings, S, find the shortest common superstring: where the set is baa, aab, bba, aba, abb, bbb, aaa, bab. This concatenates to: baaaabbbaabaabbbbbbaaabab, but the shortest common superstring is aaabbbabaa - there is no shorter string which contains all these substrings. This is at the heart of the assembly problem. However, there are some downsides to this problem - it's not tractable with no efficient algorithms for it (NP-complete) - as input strings grow, it'll slow considerably. One solution is to just pick the order for strings and then construct the superstring. However, this is extremely slow - if S contains n strings, there are n! possible orderings. We can use the overlap function from above to implement a brute force method which will be extremely slow, but correct. A faster algorithm: a 'greedy' common superstring - which chooses largest possible overlap at each point and then moves forward. This is faster than the naive approach - but at a cost - doesn't always find the correct answer - reporting a superstring which is not necessarily the shortest one.

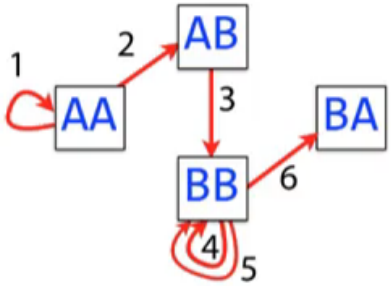
```
def pick_maximal_overlap(reads,k):
    reada, readb=None, None
    best_olen=0
    for a,b in itertools.permutations(reads,2):
        olen=overlap(a,b,min_length=k)
        if olen>best_olen:
            reada, readb = a,b
            best_olen=olen
    return reada, readb, best_olen
```

So far: we've seen greedy and brute force, but both have downsides. When the genome is repetitive, the SCS of the reads is repetitive - we can't tell exactly how many copies we have in the original sequence - the reads just aren't long enough to tell us how long it is, causing ambiguity and its not easy or possible to reassemble the genome. **Third law of assembly: repeats make assembly difficult**. You don't want to collapse them down into too few a number of repeats e.g. - 'this is a serious serious serious problem' into 'this is a serious serious problem'. About half the genome is covered by repetitive elements. A De Bruijn graph: assume that sequencing reads consist of each of the k-mers once - for each k-mer make an addition to the corresponding graph. From a De Bruijn graph, you can re-construct the original graph for walking through each graph following each edge as we go, respecting the edges: it uses each k-mer exactly once. **This is called an Eulerian walk** through the node to node to give us the reconstructed genome sequence back again.

Directed Graph Example



A Eulerian walk



However, this doesn't allow us to escape from the third law - that repeats make things difficult. However, there might be multiple Eulerian walks which creates ambiguity for sequence reads. Decreasing the length of the k-mer increases the chance of there being multiple different walks - the curse of the repetative genome. The De Brujin graph is a very common way to represent the assembly problem - used in a lot of softwares (internally) - despite the fact that the SCS and Eulerian walk are flawed formulations for the assembly problem, the overlap (overlap-layout consensus assembly - OLC) /DB (De Brujin graph assembly - DBG) graph are still going to be very useful to us in practice. How do real software tools work? In practice, the graphs are extremely messy for lots of reasons like sequencing errors or other 'dead ends'. Another reason is polyploidy. We can deal with these problems by chopping our assembly into pieces which have no ambiguity: we can still put portions of the puzzle together - these are called 'contigs': partial reconstructions which we can put together unambiguously. Therefore, an assembler reports a set of 'contigs' - even the Human Reference Genome still has holes in it, which is something we have to live with due to repetitive DNA. One solution is to make the reads longer - to glue it to some surrounding non-repetitive sequence. This is like making the puzzle pieces bigger. There doesn't exist technology which can collect reads longer than tens of thousands of bases. Paired-end 'Template' - can give us a bit more information per sequence. It gives about twice as many bases and is extremely common in practice. It doesn't sacrifice much in terms of accuracy or speed. Other technologies are getting better, but are much slower - sequencing one molecule at a time (rather than a lot in parallel) with lots of mistakes.

```
def greedy_scs(reads,k):
    read_a,read_b, olen=pick_maximal_overlap(reads,k)
    while olen>0:
         reads.remove(read_a)
         reads.remove(read_b)
         reads.append(read_a+read_b[olen:])
    return ''.join(reads)
```