CS 170 Algorithms Soln 9
Fall 2014 David Wagner

There are a variety of approaches you could use to solve this problem. Here are two:

- Greedy algorithm, based upon greedily merging the fragments with the largest overlap (see § 1).
- A random walk algorithm, based upon building a graph of the short reads (with edges indicating the amount of overlap) and then doing a random walk through the graph (see § 2).

These two approaches are explained in detail below. There are other possible approaches as well.

All of these algorithms need a way to compute the overlap between a pair of strings, so we'll describe how to do that, after summarizing the two approaches above.

We will assume we are given a set $S = \{s_1, s_2, \ldots, s_n\}$ of short reads, and that there are no duplicated short reads.

1 Greedy merging

Main idea. One way to solve this problem is to use a greedy approach. We repeatedly apply the following process: find $s,t \in S$ that have the largest overlap, merge s,t to get a new string u, remove s,t from S, and add u to S. After at most $n-1$ iterations, the set S will contain only a single string; output that string.

Here's a refinement that's helpful: before beginning the above process, we preprocess the short reads to remove any short read that is a substring of any other short read.

We get the following pseudocode:

```
1. For each $s,t \in S$ with $s \neq t$:
2.     If $s$ is a substring of $t$, remove $s$ from $S$.
3. While $|S| > 1$:
4.     Find $s,t \in S$ that maximize Overlap$(s,t)$, subject to $s \neq t$.
5.     Merge $s,t$ to get a new string $u$.
6.     Add $u$ to $S$. Remove $s,t$ from $S$.
7. Output the one string in $S$.
```

Efficient implementation. The challenging part is to provide an efficient implementation for this algorithm. If we re-calculate the overlap between all pairs in each iteration of the while loop, the running time will be $O(n^3k^2)$, which is inefficient. We can optimize this by avoiding re-doing work unnecessarily.

One good improvement is to avoid re-computing the overlap between two strings that haven't changed. We can initially compute the overlaps between all pairs of short reads in a pre-processing step. Then, each time we merge two strings, say $s,t \in S$, it is possible to compute the overlap between the new string u and every other $x \in S$ efficiently. In particular, $\text{Overlap}(u,x) = \text{Overlap}(t,x)$ and $\text{Overlap}(x,u) = \text{Overlap}(x,s)$, assuming no element of S is a substring of any other. See § 3 for the details. There is no need to recompute the overlap between x,y for any other pair of strings x,y that weren't involved in the merger.

This suggests storing the overlap information in some data structure that makes it easy to find the pair with largest overlap, and updating the data structure each time we merge two strings. One reasonable data structure is a priority queue that stores all pairs of elements of S, prioritized by their overlap. The priority queue contains $O(n^2)$ elements, so if we use a binary heap, each operation on the priority queue takes $O(\lg(n^2)) = O(\lg n)$ time.

We can reduce the space consumption with a little more cleverness: for each string $s \in S$, we remember the string $t \in S$ that maximizes $\text{Overlap}(s,t)$ (the best string to put just to the right of s) and the string $r \in S$ that maximizes $\text{Overlap}(r,s)$ (the best string to put just to the left of s). We can maintain a priority queue of these $O(n)$ pairs. The asymptotic running time for the priority queue operations remains the same, but the space efficiency is greatly improved.

How do we ensure that no element of S is a substring of any other? Lines 1–2 in the pseudocode deleted all short reads that are a substring of some other short read. Fortunately, merging cannot create any new violations of this property. In particular, we obtain the invariant that no element of S will be a substring of any other element of S, in any iteration of the while loop. (Why? Imagine merging $s,t \in S$ to get u. Could this create a new substring relationship, where some $x \in S$ is a substring of u? No. By the inductive hypothesis, x is not a substring of s or t. Therefore, if x is a substring of u, x must overlap with a suffix of s. But since the greedy algorithm merged the two strings with highest overlap, $\text{Overlap}(s,x) < \text{Overlap}(s,t)$, so x must be a substring of t, which is a contradiction.)

Running time. The greedy algorithm can be implemented in $O(n^2k^2)$ worst-case running time, using a priority queue. Pre-computing all-pairs overlaps takes $O(n^2k^2)$ time, because we invoke Overlap() for each pair of short reads. Inserting them all into the priority queue can be done in $O(n^2)$ time, if we use a binary heap (using heapify), or in $O(n)$ time (if we're using the space improvement mentioned above). We do $\leq n-1$ iterations of the while loop. Each iteration does a few priority queue operations ($O(\lg n)$ time), merges the two strings ($O(k)$ time), and updates the overlap information ($O(n)$ time, using the ideas shown above). Therefore, the total running time is $O(n^2k^2)$.

The average-case running time is $O(n^2k)$, since the average running time to compute the overlap of two random short reads is $O(k)$. See § 3 for the details.

You can find one sample implementation—Java code, and documentation for each step—in § 5.

Optimization: Speeding this up with hash tables. One risk of the greedy algorithm is that if it makes a bad choice, merging two short reads that weren't actually from adjacent positions in the original DNA sequence, this may lead us down a bad path. What can we do to reduce the likelihood of bad choices? Intuitively, the longer the overlap between two short reads s,t, the more likely it is that they were from adjacent positions in the original DNA sequence. (Two non-adjacent short reads are unlikely to happen to overlap by many characters; that's a low-probability event.)

This suggests we might want to impose a minimum threshold α, where we only consider two short reads eligible for merging if their overlap is at least α. We could continue to use the greedy algorithm, selecting at each step the pair of elements of S with largest overlap, except that we disregard any pair with overlap less than α. When we run out of possibilities to merge, we output the longest string in S. As long as there are enough short reads, this procedure should still find the original DNA sequence correctly (it is unlikely to terminate early).

Interestingly, this variant enables a major speedup to the asymptotic running time. Section 3 shows how to use hash tables to very efficiently find all pairs that have overlap $\geq \alpha$. The idea is to store the elements of S in a hash table, where the key for each string $s \in S$ is the first α letters of s. With this data structure, given a string $s \in S$, we can find all other elements of S that have overlap $\geq \alpha$ with s in $O(k)$ time on average (if we treat α as a small constant). This lets us implement the greedy algorithm with an average-case running time of $O(nk)$, plus the time to filter short reads that are a substring of some other short read; $O(n^2k)$ in total. A typical value of α might be 4 or 5 or 6.

A graph-based version. We can use the same idea, but using a graph. The following are the main steps with the corresponding running-times.

1. Preprocess by removing any short read that is a substring of another short read. This takes $O(n^2k^2)$ time.

2. Create a directed graph $G=(V,E)$ such that V is the set of vertices where each vertex represents a short read. Each directed edge (v_1,v_2) has weight equal to the maximum overlap between short read v_1 and short read v_2, where v_1 is placed to the left of v_2. Also create a priority queue to store the list of all edges, with their weight used as the priority, and add all edges to the priority queue. This takes $O(n^2k^2)$ time.

3. In each iteration, use the priority queue to find the highest-weight edge that hasn't been processed yet. ($O(\lg n)$ time)

4. If either endpoint of the edge is marked, ignore this edge and go back to the previous step. Otherwise, merge the two short reads connected by that edge (which will be implemented by adding this edge to a path that represents our solution), then mark both vertices as “used”.

5. Repeat until we have added $n-1$ edges to our solution set. This entire process takes at most $O(n^2 \lg n)$ time, since there are at most n^2 edges.

6. Start from the first vertex in our solution set (a vertex that has no incoming edges) and merge the short reads by following the corresponding edges. Return the final output. This takes $O(n)$ time.

The overall running time of this algorithm is $O(n^2 k^2 \lg n)$. It can be reduced to $O(n^2 k^2)$ running time by using an appropriate priority queue data structure: since all of the priorities (edge weights) are in the range $0..k$, we can use an array $A[0..k]$ where $A[i]$ is a list of all edges with weight i, and then each priority queue operation takes $O(1)$ amortized time.

The average-case running time is $O(n^2 k \lg n)$ or $O(n^2 k)$, depending upon which priority queue implementation we use.

2 Random walk

Main idea. Another approach is to form a graph to record the overlap information, and then do a random walk through this graph. Form a graph where each short read is a vertex, and where we have an edge from s to t whose weight is given by $\text{Overlap}(s,t)$. We can do a random walk through this graph, starting at some vertex. When we are at vertex v, we randomly choose an edge out of v to follow, in a way that is biased towards edges with high weight.

Each path defines a candidate reconstruction for the DNA sequence: put the short reads in the order that they are visited in the path, and then merge them. We can repeat this process many times and choose the best solution found.

We get an algorithm like this:

```
1. Compute Overlap$(s,t)$ for all $s,t \in S$ and build the overlap graph.
2. Repeat 100 times:
3.     Choose a starting vertex $v_0$ somehow.
4.     Choose a random path starting at $v_0$, where at each step the random choice is biased towards
       edges with large weight.
5.     Obtain a corresponding reconstruction, by merging short reads in the order described by the path.
6. Output the best reconstruction found.
```

There are many details that must be filled in, to get a full algorithm; let's look at some potential approaches.

Random path selection. We can construct a random path iteratively: when we are at vertex v, we look at the edges out of v and randomly select one of them to follow, according to some probability distribution on the edges. One good strategy is to make the probability of following the edge (v,w) be proportional to $4^{\text{weight}(u,v)}$, i.e., proportional to $4^{\text{Overlap}(v,w)}$. This reflects the fact that the probability of a false match (where v,w weren't actually adjacent in the original DNA sequence but happen to end up overlapping, by random chance) is proportional to $1/4^{\text{Overlap}(u,v)}$, so edges with a large weight are less likely to represent false matches and should be chosen with higher probability.

Starting point. We need to choose a vertex to start at. One approach is choose a short read v such that $\text{Overlap}(s,v)$ is small for all $s \in S$. For instance, we can define $f(v) = \max\{\text{Overlap}(s,v) : s \in S\}$ and let v_0 be the vertex that minimizes $f(v_0)$. Or, we can randomly choose a vertex with a distribution that is heavily weighted towards vertices whose $f(\cdot)$-value is small.

Termination condition. We need to decide when to stop the random walk and end the path. One possible termination condition is that if we reach a vertex (short read) v such that $\text{Overlap}(v,w)$ is small (e.g., ≤ 4) for all $w \in S$, then we stop the walk. Alternatively, we can stop once every short read $s \in S$ is a substring of the reconstructed string: we keep track of the reconstructed string that corresponds to the path so far, and we keep track of the short reads that haven't been covered yet (that aren't a substring of the reconstructed string so far). Each time we extend the path by one vertex, we update the reconstructed string, and remove each short read that has been newly covered. We can use the overlap information to help automate the latter check.

Measuring the quality of the reconstruction. To choose the best reconstruction, we need a way to measure how good each candidate reconstruction is. There are several different possibilities. One metric is the length of the reconstructed string. Another metric is the number of short reads it covers (i.e., the number of short reads that are a substring of it). Better still might be to use some weighted average of those two metrics.

Pre-processing: filtering out redundant short reads. Empirically, this algorithm seems to work better when we filter out redundant reads: in a pre-processing step, check all pairs of short reads and remove any short read that is a substring of some other short string.

Running time. With suitable data structures, this can be implemented with worst-case runtime $O(n^2k^2)$ and average-case runtime $O(n^2k)$.

If we establish a threshold α and only consider overlaps of at least α characters, and we use the hash-based optimization mentions in Sections 1 and 3, we can get the average running time down to $O(nk)$ plus the time to filter short reads that are a substring of some other short read; $O(n^2k)$ in total.

3 Computing the overlap between two strings

Problem statement. Suppose we have two strings u, v. How can we efficiently compute the maximum overlap between these two strings, assuming u is on the left and v is on the right?

In other words, suppose the length of u is ℓ_u, and the length of v is ℓ_v. We want to find the largest number m such that $u[\ell_u - m + 1..\ell_u] = v[1..m]$. (Then we can merge them into a string of length $\ell_u + \ell_v - m$.)

Straightforward solution. The simple solution is this: we try each possible value of m, starting with the largest possible value of m down to one. In other words:

```
Overlap($u, v$):
1. For $m := \min(\ell_u, \ell_v), \ldots, 2, 1$:
2.     If $u[\ell_u - m + 1..\ell_u] = v[1..m]$, return $m$.
3. Return 0 (no overlap).
```

Notice that line 2 can be done in $O(m)$ time (in the worst case, we need to iterate through m characters of u, v). If both strings are of length $\leq k$, then the worst-case running time to compute the overlap is $\Theta(k^2)$.

Of course, if we don't care about which string appears first, we can compute Overlap(u, v) and Overlap(v, u).

Average-case running time. If u, v are randomly generated strings, the expected running time (average-case running time) of Overlap(u, v) is $\Theta(k)$.

Why? Suppose strings s, t are randomly generated: each character is uniformly and independently distributed on $\{A, C, G, T\}$. Let the random variable X denote the smallest value of i such that $s[i] \neq t[i]$, i.e., the first character that differ in s, t. Then X has a geometric distribution with parameter $p = 3/4$, namely, $\Pr[X = i] = 3/4^{i+1}$, so using standard facts about the geometric distribution, $\mathbb{E}[X] = 1/p = 4/3$, which is a constant. Letting $s = u[\ell_u - m + 1..\ell_u]$ and $t = v[1..m]$, we see that the expected running time for line 2 of the algorithm is $\Theta(1)$. There are at most k iterations of the loop, so the total expected running time for Overlap is $\Theta(k)$.

Finding all pairs with large overlap. Some of our algorithms will seek to find all pairs with an overlap larger than some threshold, say α. There is a straightforward algorithm with $O(n^2k^2)$ worst-case running time and $O(n^2k)$ expected running time: just try all pairs, as shown below.

```
PairsWithLargeOverlap():
1. For each $s, t \in S$ such that $s \neq t$:
2.     If Overlap$(s, t) \geq \alpha$, output the pair $(s, t)$ and continue.
```

Optimization: hash tables. We can speed up the process of finding all pairs with large overlap using a hash table. In particular, we can build a hash table that stores each short read s, keyed on $s[1..\alpha]$ (the first α letters of s). This hash table can be built in $\Theta(n\alpha)$ time. Once we have the hash table, then given a short read s we can quickly find all short reads t that have a large overlap with s as follows:

```
Successors($s$):
1. For each $i := 1, 2, \ldots, \text{length}(s) - \alpha + 1$:
```

```
2.        Look up $s[i..i+\alpha-1]$ in the hash table, output all $t \in S$ such that $t[1..\alpha] = s[i..i+\alpha-1]$, and continue.
```

```
PairsWithLargeOverlap():
1.  For each $s \in S$:
2.        Output Successors($s$) and continue.
```

The expected running time of this procedure will be $O(nk\alpha)$, if α is large enough.

Why? Well, we can construct the hash table in $\Theta(n\alpha)$ time. Each call to Successors(s) takes $\Theta(\alpha k + n_s)$ time, where n_s is the number of strings it outputs. Therefore, the overall running time of PairsWith-LargeOverlaps is $\Theta(nk\alpha + N)$ where N is the total number of pairs with overlap $\geq \alpha$. We'll argue that if α is large enough, than N is not too large. Assuming no short read is duplicated, there will be $\leq nk$ pairs of short reads that are chosen from overlapping positions in the original string. Also, if we have two short reads s,t that aren't chosen from overlapping positions in the original string, the probability that their overlap is $\geq \alpha$ will be $\leq k/4^\alpha$ (use a union bound, and note that the probability of an overlap of $\geq \alpha$ letters starting at any fixed location is $1/4^\alpha$). Therefore, if we choose $\alpha \geq \log_4 n$, the expected number of pairs with overlaps $\geq \alpha$ will be at most $2nk$, i.e., $\mathbb{E}[N] \leq 2nk$. Therefore, the expected running time of PairsWithLargeOverlaps will be $O(nk\alpha)$, in total.

This also shows how to efficiently find all strings $t \in S$ such that Overlap$(s,t) \geq \alpha$, for a given s: we simply call Successors(s) above. The average running time of Successors(s) is $O(\alpha k + n_s)$ where n_s is the number of strings it outputs; on average, n_s is about $2k$, so the average running time is $O(\alpha k)$.

It is possible to further improve the running times to eliminate the factor of α, by using a *rolling hash*. Normally, computing the hash of $s[i..i+\alpha-1]$ takes $\Theta(\alpha)$ time, since you must look at all α characters in $s[i..i+\alpha-1]$. A rolling hash has the following nifty property: once you've computed the hash of $s[i..i+\alpha-1]$, you can compute the hash of $s[i+1..i+\alpha]$ in $O(1)$ time. (See, e.g., `https://en.wikipedia.org/wiki/Rolling_hash`.) With this improvement, Successors(s) takes $O(k+\alpha)$ time on average and finding all pairs with overlap $\geq \alpha$ takes $O(n(k+\alpha))$ time, on average. However, if α is small, the improvement may be modest.

Optimization: Calculating the overlap after merging two strings. Suppose that we've already calculated Overlap(s,x) and Overlap(t,x) for all $x \in S$, and we merge s and t to get the string u. How efficiently can we compute Overlap(u,x) for all $x \in S$?

It turns out it is possible to save work. Let's make the simplifying assumption that no element of S is a substring of any other (e.g., no short read is a substring of any other short read). Then it follows that Overlap(u,x) = Overlap(t,x). (If the longest overlap in u,x started before the beginning of t, then t would be a substring of x, contradicting our assumption.) This means there is no need to re-compute the overlap from scratch.

One way to ensure the simplifying assumption holds is to pre-process all the short reads to remove any that are a substring of some other short read. The pre-processing can be done in $O(n^2k^2)$ worst-case time or $O(n^2k)$ expected time by iterating over all pairs $s,t \in S$. For some algorithms, including our greedy algorithm, this pre-processing is all you need to do.

For other algorithms, whenever we merge two strings in S, we might need to remove all other elements of S that are a substring of the merged one. Fortunately, there is a trick that lets us do this efficiently if we have already calculated the overlap between all pairs of strings in S: when we merge two strings s,t to get u, we can efficiently detect whether there is some $x \in S$ that is a substring of u using our previously computed overlap information. If x is a substring of u, then it must overlap both s and t. So, Overlap(s,x) tells us where the start of x appears in u and $\text{Overlap}(x,t)$ tells us where x ends in u. In particular, x is a substring of u if and only if $\text{Overlap}(s,x) + \text{Overlap}(x,t) = \text{len}(x) - \text{Overlap}(s,t)$. However, this last trick is not necessary for the greedy algorithms shown next.

4 Advanced techniques

There are some advanced techniques available that are beyond the scope of what I'd expect anyone to try in this class, but I thought I'd share them in case they are of interest.

Suffix trees. Suffix trees can be used to speed up these algorithms. You can preprocess the short reads to remove any short read that is a substring of some other short read in $O(nk)$ time, using a suffix tree. Also, given a short read s, you can efficiently find all $t \in S$ such that $\text{Overlap}(s,t) \geq \alpha$, using a suffix tree: build a (generalized) suffix tree for the reversals of all the strings in S, then look up s in it. For instance, this can be used to implement the greedy algorithm and the random-walk algorithm in $O(nk)$ running time, which is very impressive. However, suffix trees are notoriously tricky to implement correct, so this is beyond the scope of what I'd expect anyone to try in this class.

Improvements to greedy. The greedy algorithm outlined in § 1 merges the two strings with largest overlap. Of course, a bad choice early on might eliminate good choices later on. There is some evidence in the research literature that we can improve on the basic greedy algorithm by using a slightly different metric to choose which pair to merge.

One metric that has been suggested in the research literature is

$$g(s,t) = 2.5 \times \text{Overlap}(s,t) - \max\{\text{Overlap}(x,t) : x \in S, x \neq s\} - \max\{\text{Overlap}(s,y) : y \in S, y \neq t\}.$$

Then, in each iteration, we merge the pair s,t that has the highest value of $g(s,t)$. The idea is that g takes into account the pairs that would be eliminated if s,t were merged: if merging s,t eliminates the possibility of later merging s,y where s,y has a pretty high overlap, then maybe we should slightly downgrade the apparent value of merging s,t.

Another metric that has been suggested in the literature is

$$h(s,t) = \min_{u} \frac{\text{length}(u)}{\sum_{x \in S, x \text{ substring of } u} \text{length}(x)},$$

where u ranges over all ways of merging the two strings s,t (not just the shortest/optimal way to merge them). For instance, if $s = ACGTG$ and $t = GTGGC$, there are three candidate values of u: $ACGTGGTGGC$, $ACGTGTGGC$, and $ACGTGGC$, corresponding to whether we overlap them by 0, 1, or 3 positions. Then, as in the basic greedy algorithm, in each iteration we merge the pair s,t that has the highest value of $h(s,t)$. The idea of this metric is to try to find two strings s,t that when merged make many other strings become a substring of the merger.

A different variation on greedy is to filter the set of pairs we consider, to try to reduce the likelihood that we make a mistake (i.e., where we merge two strings that weren't adjacent in the original sequence). If we have two strings $s,t \in S$ that we're considering merging, call string $x \in S$ a *certificate* if s,x has non-zero overlap and x,t has non-zero overlap. Intuitively, the existence of such a certificate makes it less likely that it is a mistake to merge s,t. Then, we can treat s,t as eligible for merging only if there is some certificate x: in each iteration, we find a certified pair whose overlap is as large as possible and merge them. In particular, only certified pairs are considered eligible for merging. Apparently, it might be even better to run the basic greedy algorithm for a few iterations, until $|S| \leq c$ (where c is some constant), and then switch to only considering certified mergers after that point.

Set cover. There is another algorithm, based on the greedy approximation algorithm for set cover that we saw earlier in this class. Let U denote the set of strings that can be obtained by merging some pair of short reads of S. For each $u \in U$, define a set

$$S_u = \{v \in S : v \text{ is a substring of } u\}.$$

The sets S_u for $u \in U$ form a collection of sets. This gives us an instance of the set-cover problem: we want to select some of these sets, so that their union is all of S.

By assigning a weight to each set, we obtain an instance of the weighted set-cover problem: select some of these sets, so that their union is all of S and the total weight of the selected sets is as small as possible.

We use $\text{length}(u)$ as the weight for the set S_u. Now look for a minimal-weight set cover, or some approximation to it. Suppose it selects sets $S_{u_1}, S_{u_2}, \ldots, S_{u_m}$ whose union is all of S. Then, we can merge the strings $u_1, u_2, \ldots, u_m$ to get a single long string, which we will treat as our candidate for the reconstructed DNA sequence. The length of this reconstruction will be at most $\text{length}(u_1) + \cdots + \text{length}(u_m) \leq \text{wt}(S_{u_1}) + \cdots + \text{wt}(S_{u_m})$, i.e., its length will be at most the total weight of the cover. One can show that the weight of the minimal-weight cover will be at most twice the length of the original DNA sequence. Also, we can use the greedy heuristic for weighted set cover to get a set cover whose weight is at most $\log n$ times as much as the minimal-weight cover. This gives us an algorithm that will output a string that contains every short read as a substring, and is at most $2\log(n)$ times as long as the original DNA sequence.

Worst-case hardness. It is known that solving the shotgun sequencing problem exactly is NP-hard: given a set S of short reads, it is NP-hard to find the shortest string that is a superstring of each short read in S. However, this relates to the worst-case hardness, rather than the average case (e.g., when the original string is randomly generated).

Approximation results. Researchers have proven that the greedy algorithm is guaranteed to output a string that is at most $3.5\times$ as long as the shortest valid solution.

There are other schemes that provably achieve a $2.5\times$ approximation ratio. Some of those schemes work by using max-flow algorithms to find a maximal matching (a collection of edges such that no two edges share a common endpoint) for the overlap graph, and then merging the two short reads that are connected by each edge. However, the details are fairly intricate, and they are slower than the greedy algorithm.

Of course, these algorithms often do even better than the provable approximation bounds would suggest, but here we are concerned only with what we can prove will hold for all possible inputs.

It is an open problem what the best achievable approximation ratio is. It is known that there is no polynomial-time algorithm that can achieve an approximation ratio of 1.0008 or better (assuming $P \neq NP$). However, this leaves a gap between the lower bound of 1.0008 and the upper bound of 2.5, and the exact best approximation ratio that's achievable in polynomial time is not known.

More reading. The following book chapter has a nice overview of the research literature on this problem: The Shortest Superstring Problem, Theodoros P. Gevezes and Leonidas S. Pitsoulis, Optimization in Science and Engineering, pp.189–227, 2014, `http://link.springer.com/chapter/10.1007/978-1-4939-0808-0_10`.

Challenges in practice. We have focused on the shortest superstring problem: given a set of short reads, find the shortest string that has every short read as a substring of it. This captures some of the algorithmic challenges in shotgun sequencing. However, in practice DNA sequencing introduces other challenges as well:

- **Errors.** In practice, some errors occur when reading DNA. Thus, each letter in each short read has some probability to be erroneous, e.g., maybe a 1% or 2% chance of being read incorrectly. This means that we need to consider approximate matches, where s,t can be overlapped in a way that makes them mostly match up in the region where they overlap (e.g., the overlapping region has a low edit distance).

- **Scale.** The data sets we gave you had thousands of short reads. In practice, the DNA sequence for the human genome is billions of letters long, and we may have hundreds of millions of short reads. Very efficient algorithms are needed to be able to handle data at this scale.

- **Repeats.** The human genome contains long repeated sequences that occur far apart from each other. This can make reconstruction much harder. For instance, suppose that the original DNA has the form $WRYRZ$, where W,R,Y,Z are long strings and R is a repeated region. Then the greedy algorithm can easily perform poorly: it might produce the reconstruction WRZ or something else. There are algorithmic techniques to deal with this, based on heuristics for finding Hamiltonian paths in the overlap graph, but this introduces a significant additional challenge.

  As another example, suppose that the DNA sequence has a long stretch of all A's that's much longer than k. Then no number of short reads of length k will suffice to uniquely determine the length of this stretch of A's. The one saving grace is that really bad corner cases like this are unlikely to occur in real DNA: DNA mutates at some low rate, so if such a DNA sequence ever existed, it is unlikely that it would have survived eons of random mutations.

If you enjoyed this stuff, you might enjoy CS 176!

5 Code for Algorithm 1

Implementation 1.

```
public static String greedySolution(List<String> list) {
    // Step 1: Remove any short read that can be contained in another short read.
    int currentIndex = 0;
    outerloop1:
    while (currentIndex < list.size()) {
        int innerIndex = currentIndex + 1;
        while (innerIndex < list.size()) {
            if (list.get(currentIndex).contains(list.get(innerIndex))) {
                list.remove(innerIndex);
            } else if (list.get(innerIndex).contains(list.get(currentIndex))) {
                list.remove(currentIndex);
                continue outerloop1;
            }
            innerIndex++;
        }
    currentIndex++;
    }

    // Step 2: Create and initialize indexMap that maps each short read to an index.
    HashMap<String, Integer> indexMap = new HashMap<String, Integer>();
    for (int i = 0; i < list.size(); i++) {
        indexMap.put(list.get(i), i);
    }

    // Step 3: Create and compute overlapMap that maps the maximum overlap between
    // two short reads to the list of strings created in following way ”ShortRead1,ShortRead2”.
    HashMap<Integer, List<String>> overlapMap = new HashMap<Integer, List<String>>();
    for (int i = 0; i < list.size(); i++) {
        for (int j = 0; j < list.size(); j++) {
            if (i != j) {
                int overlap = findLargestOverlap(list.get(i), list.get(j));
                if (!overlapMap.containsKey(overlap)) {
                   overlapMap.put(overlap, new LinkedList<String>());
                }
                overlapMap.get(overlap).add(list.get(i) + ”,” + list.get(j));
            }
        }
    }

    // Step 4: Create and initialize mergingOrderLeftArray and mergingOrderRightArray.
    // mergingOrderLeftArray[i] = -1 if ith index short read has not been merged on its left side and
    // mergingOrderLeftArray[i] = j if ith index short read has not been merged on its left side with
    // jth short read. mergingOrderRightArray is the same idea but on the right side.
    int[] mergingOrderLeftArray = new int[list.size()];
```

```
int[] mergingOrderRightArray = new int[list.size()];
for (int i = 0; i < list.size(); i++) {
    mergingOrderLeftArray[i] = -1;
    mergingOrderRightArray[i] = -1;
}

// Step 5: Get the list of keys to the overlapMap in sorted order from largest to smallest.
Set<Integer> overlapSet = overlapMap.keySet();
List<Integer> overlapList = new LinkedList<Integer>();
overlapList.addAll(overlapSet);
Collections.sort(overlapList, Collections.reverseOrder());

// Step 6: Find the merging order for all short reads.
int mergeCount = 0;
outerloop2:
for (int k : overlapList) {
    List<String> currentList = overlapMap.get(k);
    for (String pair : currentList) {
        String[] strings = pair.split(",");
        int index1 = indexMap.get(strings[0]);
        int index2 = indexMap.get(strings[1]);
        if (mergingOrderRightArray[index1] == -1 && mergingOrderLeftArray[index2] == -1) {
            mergingOrderRightArray[index1] = index2;
            mergingOrderLeftArray[index2] = index1;
            mergeCount++;
        }
        // If we have merged n - 1 times, quit merging.
        if (mergeCount == mergingOrderRightArray.length - 1) {
            break outerloop2;
        }
    }
}

// Step 7: Get the first short read, the short read that should not be merged on the left.
int index = 0;
for (int i = 0; i < mergingOrderLeftArray.length; i++) {
    if (mergingOrderLeftArray[i] == -1) {
        index = i;
    }
}

// Step 8: Merge the short reads in order we computed to get the final DNA.
String dna = list.get(index);
while (mergingOrderRightArray[index] != -1) {
    int nextIndex = mergingOrderRightArray[index];
    dna = dna + list.get(nextIndex).substring(findLargestOverlap(list.get(index), list.get(nextIndex)));
    index = nextIndex;
}
```

```
    return dna;
}

private static int findLargestOverlap(String x, String y) {
    int length = Math.min(x.length(), y.length());
    for (int l = length - 1; l > 0; l-=1) {
        if (checkOverlap(x, y, l)) {
            return l;
        }
    }
    return 0;
}

private static boolean checkOverlap(String x, String y, int overlap) {
    for (int i = 0; i < overlap; i++) {
        if (x.charAt(x.length() - overlap + i) != y.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

Running-time analysis. The asymptotic running-time of the algorithm is $O(n^2k^2)$. First, we take note that the method checkOverlap runs in $O(k)$ time given strings of length $O(k)$. Therefore, the findLargestOverlap method runs in $O(k^2)$ time given strings of length $O(k)$. Now, we will analyze the running-time of the greedySolution method. We will analyze step by step.

1. Checking each short read with every other short read takes at maximum $O(n^2)$ iterations. To check if one short read is contained in another, we use a naive approach of checking at each index of the longer string, whether or not all characters of the smaller string match each character. We can compute this in $O(k^2)$ so this process takes $O(n^2k^2)$.

2. Creating and initializing the index map takes $O(n^2)$ since we are just iterating through the list of short reads.

3. To compute the overlapMap, we take each possible pair of short reads and compute the largest overlap between them. We know there are $O(n^2)$ possible pairs of short reads given n short reads and computing largest overlap takes $O(k^2)$ time. Creating a list and adding an element to the list takes $O(1)$ time. Therefore, this takes $O(n^2k^2)$ time.

4. This step takes $O(n)$ time since we are creating arrays of size $O(n)$.

5. Notice, that the largest overlap cannot be greater than or equal to k since k is the length of the longest short read. Therefore, sorting keys that are at most k, takes $O(k \log k)$.

6. We know that if we combine the length of all lists in the map overlapMap, this sum is in $O(n^2)$ (each element in list represents a pair of strings and it's largest overlap). Therefore, finding the merging order takes $O(n^2)$.

7. Finding an element in array of length $O(n)$ takes $O(n)$ time.

8. This step takes $O(n)$ time since this while loop is guaranteed to end after $n-1$ iterations.

By summing up the asymptotic running-time of all the steps, we get $O(n^2k^2)$.