Explanation of Algorithm

Each short can be represented as a node on a directed graph, with an edge AB representing that A is a suffix of B, with the weight of edge AB representing the number of letters which overlap. Finding all possible edges in the graph (corresponding to our list of possible pairs of shorts) thus takes N^2 string comparisons, each of which might take up to K time. In practice, the data structure used throughout the code represents a path on this graph and takes the form [start node, end node, "string", weight, set of nodes in path]. This fulfilled sort of specific graph manipulations required, without the overhead of building an explicit graph.

Given this graph, we want to find the highest weighted path which includes every node; this is essentially a slight variation on the traveling salesman problem. However, because the samples we are given are randomized, the probability of incorrect edges decreases exponentially with the weight of the edge. Thus a greedy solution is very likely to be a correct solution. I initially tries greedily grabbing edges with the most weight until each original short was accounted for, but this didn't work. Rather, in order to ensure that all the edges fit together to cover all original shorts, it is necessary to limit possible next edges to children of the nodes/Shorts at the beginning and end of our accumulated pairing, filtering out those edges which connect to an already visited node. The graph problem is essentially a traveling salesman problem (attempting to find a path which visits all nodes using each edge at most once and using the highest weighted edges) and our greedy solution is fairly similar to the standard greedy (sub-optimal) solution for TSPs. Forcing consecutive pairings also allows the strings to be merged as we progress through the sorting, saving post-processing and allowing for overlaps which might be larger than a single element in the pairing.

To demonstrate how this greedy solution works, let's call any observed overlap between pairs, which does not belong in the final sequence, a 'rogue overlap'. With randomly chosen letters, the chance of a rogue overlap of length j is $(.25)^j$. This means that we can assume with high probability that, when choosing between two overlaps to include in our final solution, the larger overlap is not a rogue overlap.

However, It's important to note that several edge cases make this greedy solution fail. For example:

cagagag
agagagc
gtttttttt

would fail, because the first two would match leaving the last in the dust.

This is an important problem in real DNA sequencing, where long repeated sequences are common. With random letters, however, the chances of such a structure occurring are negligible, as the algorithm will stop appending nodes to the graph after all initial shorts have been accounted for, and real overlaps will be much larger than random overlaps.

Pseudocode

Step 1: Pre-Process-

- Remove all "sandwiched" shorts (shorts which are entirely contained within another short) This takes kN^2, as each of N shorts must be compared to N-1 other shorts, in a K time string comparison

- Generate all possible pairs of shorts. For each short, look at every other short. consider the first as a prefix and the second as a suffix. Iterate up to the length of the shortest, seeing if the last i letters of the prefix are the same as the first K letters of the suffix. If there's a match, we store this as an edge of the form [prefix index, suffix index, merged string, overlap weight, prefix index, suffix index]. N^2 comparisons each take K time, so this is also $O(kN^2)$

Step 2: Greedily built path

- sort pairs by descending overlap weight. While the mechanics of this sort are hidden by python abstraction, I assume it takes about nlogn time.

- Take the first pair, which has the greatest overlap. This current pair will be built up into our result. We find all 'forwards' (pairs which begin with the suffix of our current pair) and 'backs' (which end in the prefix of the current pair), and find the best of each. Each of these requires examining each pair, and there could be up to N^2 pairs, but no string comparison is required.

- While there remain shorts unnaccounted for in our "visited" set (stored in our result path) add the greater of the best forward or the best back to the front or back of the result path, respectively (this involves merging the string onto the result string, re-setting the forwardmost or backmost node of that string, and adding nodes in the pair to the visited set contained in the result path). Then, refilter sortedpairs to find a new list of potential forward children (if forward was chosen) or backward children (if back was chosen). Note that the first item of these filtered lists will still be the item with the greatest overlap weight, as the list was already sorted. This process will require slightly less than N iterations (sandwiched shorts are ignored). In each iteration, one filtering action takes place (which requires N^2 comparisons) and one merge action takes place (which requires up to 2K character comparisons). Overall, this step takes $N(K + N^2)$ time.

-Once there remain no unaccounted for shorts, return the mergestring contained in the result path.

Running Time

Recall that the running time for each step in the algorithm was:

- KN^2 for removing sandwiches
- KN^2 for finding pairs
- $NLogN$ for sorting
- $2N^2$ for initial filtering
- $N(K + N^2) = NK + N^3$ for greedy grabbing of best connected edges

(These runtimes are justified in the pseudocode section).

$2N^2$, NK, and NLogN are all dominated by kN^2 running time, as is so running time is order $kN^2 + N^3$. Thus, we can say that generally the runtime is $O(KN^2 + N^3)$, but if the number of shorts is greater than the length of the longest short, running time is $O(N^3)$ and otherwise the running time is $O(KN^2)$.