**Algorithm:**
The approach I chose to use for this problem was to use a greedy solution. The algorithm will compare each string to every other string and record the amount of overlap each string has to every other string. After this calculation is done, the algorithm will then pick the two strings with the largest overlap and join the two strings to make a combined string. The original strings will be removed from the list and this new combined string will be added on. The process repeats again until there is only one string left in the initial list. That string will represent the final combination of all the short reads or the DNA sequence.

In my opinion, I feel that there is definitely a better algorithm than this to solve this problem. For instance, I tried to solve the overlap matches using dynamic programming but that caused the program to find the longest substring instead of the longest overlap. The difference is the algorithm should only care about the two strings matching at their ends. For instance, if a substring AGGA existed in TCAGGACT and in another string, CTAGGATC, there is no overlap and this calculation should be avoided. Even then, the algorithm still remains greedy to dig out the final substring so in other words, this was only a slight optimization attempt for the greedy algorithm.

Another optimization attempt was to do a full calculation of the overlaps initially. Then, the algorithm will find the biggest overlap but will only recalculate the overlap values that overlapped with the two original strings needed to create the combined string. For instance, if the short reads consisted of 6 strings [A … F] and the biggest overlap occurred between B and C. Let's say B also overlapped with A and C also overlapped with D. The only recalculations that would be performed would be for the combined string B + C with short read A and short read D. All the other overlap calculations will remain the same. This significantly reduces the amount of overlap calculations needed and overlap values can be stored in a dictionary for constant time access, instead, or a linear calculation procedure.

**Pseudocode:**
```
Algorithm SequencingDNA(list short_reads):
    while length of short_reads > 1:
        /* values represents a tuple of indices related to short_reads */
        values = max_overlaps(short_reads)
        seq1 = item in short_reads at values[0]
        seq2 = item in short_reads at values[1]
        combined = combine seq1 and seq2
        set item in short_reads at values[0] to combined
        remove item in short_reads at values[1]
    /* After all calculations and reductions are done, the final output
     * should lie at the beginning of the short_reads */
    return short_reads[0]

def max_overlaps(list short_reads):
    for (i = 0; i < len(short_reads); i++)
        for (j = 0: j < len(short_reads): j++)
            seq1 = short_reads[i]
            seq2 = short_reads[j]
            result = find biggest overlap between seq1 and seq2
    return result
```

**Efficiency and Runtime:**
As stated in the algorithm, the efficiency and runtime of this approach isn't the best but the best I could implement and get correct data values from. The runtime for this would be at O(kn^3) because of all the checks required find the longest overlap. This is the runtime since in the pseudocode, the algorithm starts off with a while loop which essentially will go through all the short_read values. This is linear since only one short_reads value is removed at a time. Then inside of this while loop, the max_overlaps takes O(kn^2) time to calculate every string to every other string and get all the overlap values. This is because the comparison between each string to every other string is already O(n^2) and each check, meaning, the process of matching the ends of the two strings, would be O(k), bumping up the whole function to O(kn^2). This is done under a linear while loop which, bumps it up to O(kn^3).

With memorization after the first initial calculation of the overlaps. The initial calculation would be the most expensive but every access after that, the calculation would perform better some of calculations can be simply retrieved from a hash which would be O(1). This optimizes the algorithm and would get it at an average runtime of Θ(n^3) if a good amount of the calculations can be simply retrieved from the hash.