Explanation: Define overlap(s, t) = the length of the longest substring which is suffix of s and prefix of t. Then every iteration, greedily merge the two strings which have largest overlap. Use a binary max heap to keep track of all overlaps left. Before merging, remove all strings which are substring of others.

Pseudocode:

```
algorithm short_reads_reconstruct(s_list):
    i_set = set(range(0, len(s_list)))
    Remove all strings which are others' substring from i_set
    for j in i_set:
        for k in i_set, k≠j:
            push (overlap(s_list[j], s_list[k]), j, k) to heap
    while len(i_set) > 1:
        (max_common, max_index1, max_index2) = pop(from heap) the largest element
        Merge s_list[max_index1] and s_list[max_index2] and store in s_list[max_index1]
        Remove max_index2 from i_set
        for j in i_set:
            if j ≠ max_index1:
                change overlap(s_list[max_index1], s_list[j]) in heap to overlap(s_list[max_index2], s_list[j])
                Delete overlap(s_list[j], s_list[max_index2])
        Delete ovelap(s_list[max_index2], s_list[j])
    Return s_list[only element in i_set]
```

Running time: $O(n^2k^2 + n^2\log n)$

Calculate all overlap values and push to heap takes $O(n^2k^2 + n^2\log n)$

Delete all strings which are substrings of others takes $O(n^2)$

There are $O(n)$ iterations, each of which takes $O(\log n)$ to pop, $O(k)$ to merge, $O(1)$ to remove from i_set, and $n \cdot O(\log n) = O(n\log n)$ to delete + change overlap

Thus, the algorithm takes $O(n^2k^2 + n^2 \log n)$ time.