Analysis – Unrestricted Algorithm:

String size n, string size m

The unrestricted algorithm starts by initializing an array of size n+1 \* m+1, (which is O(n+1 \* m+1), but as n and m get bigger and bigger the +1 becomes so small in comparison that we can simplify to O(nm)) which takes O(nm) time and O(nm) space. This is the only substantial space that we initialize in the algorithm, as all other parts of our algorithm only take O(1) space when making a single variable. Later, we concatenate the sequences as strings, which takes O(m) or O(n) space, depending on which input is longer as our ending strings will be length of the longest string. This is small in comparison to O(nm) space of the 2d array, so we can ignore this.

Apart from initializing the 2d array, we then loop through each element, each taking O(1) time to determine the minimum value from its neighbors, which leaves us with n\*m elements each with O(1) time. This leaves us with another O(nm) time chunk, but when added together to get O(2nm), we simplify to O(nm). When concatenating the strings at the end, our time complexity is O(m) or O(n), as we traverse all the way to the origin of our 2d matrix of size n\*m. By comparison, these times are small when compared to O(nm), so we are left with only O(nm) time complexity and O(nm) space complexity.

See code for more analysis on each function’s individual time and complexity

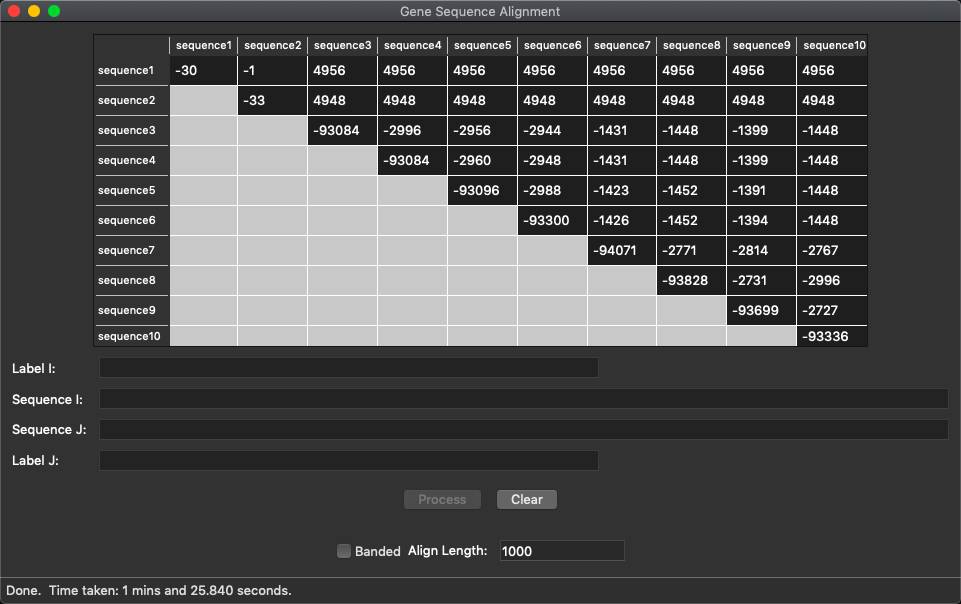
Analysis – Banded Algorithm:

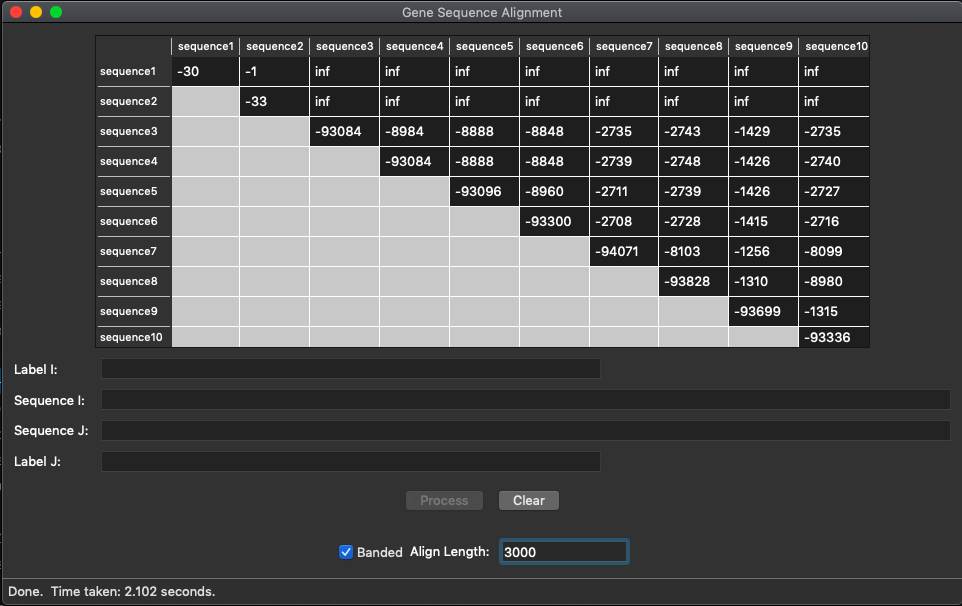
String size n, string size m, size k bandwidth

The unrestricted algorithm starts by initializing an array of size k\*n, where n is the longest string which takes O(kn) time and O(kn) space. This is the only substantial space that we initialize in the algorithm, as all other parts of our algorithm only take O(1) space when making a single variable. Later, we concatenate the sequences as strings, which takes O(m) or O(n) space, depending on which input is longer as our ending strings will be length of the longest string. This is small in comparison to O(kn) space of the 2d array, so we can ignore this.

Apart from initializing the 2d array, we then loop through each element, each taking O(1) time to determine the minimum value from its neighbors, which leaves us with k\*n elements each with O(1) time. Because we only care about the elements that are within d=3 elements from the diagonal, we only need to iterate through 7 elements on each row of the matrix, so even though we are comparing each element of n and m, we only do this when m is +- 3 from the diagonal, which leaves us iterating through k\*n elements. This leaves us with another O(kn) time chunk, but when added together to get O(2kn), we simplify to O(kn). When concatenating the strings at the end, our time complexity is either O(m) or O(n), as we are traversing our 2d matrix of size k\*n all the way back to the origin point. By comparison, these times are small when compared to O(kn), so we are left with only O(kn) time complexity and O(kn) space complexity.

How it works:

Results - Screenshots:



Results - Alignment:

K = 1000

gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a

-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

K = 3000

gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a

-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

Commented Code:

Unbanded Algorithm:

Banded Algorithm: