**Project 4: Gene Sequencing**

## Algorithm Analysis

1. Unbanded Needleman-Wunsch **(Functional)**
    a. Runtime Complexity - **O(n*m)** where n is the length of the first sequence and m is the length of the second sequence.
        i. **[1]**(n * m + n + m) + **[2]**(½)(m - 1)(n - 1) + **[3]**(m+n)
            1. The first item above is the runtime complexity of initializing the result matrix with the base cases filled in.
            2. The second item is the complexity of filling out the standard case for the rest of the matrix. However, since we only fill the upper diagonal of the matrix, we can ignore roughly half of the work performed in this section.
            3. The third item is the worst case complexity of returning the proper path. The absolute worst it could be is straight up the right side, and straight left across the top until [0][0].
        ii. The full runtime simplifies to roughly 1.5(m*n), or **O(m*n)**. The line-by-line breakdown can be found in the method body of the submitted code - **unbanded_needleman().**
    b. Memory Complexity - Unbanded, my implementation initializes the full m*n matrix, so at the absolute worst, **O(m*n)** memory is used.

2. Banded Needleman-Wunsch **(K*N Matrix - Semi-Functional)**
    a. Runtime Complexity - O(k*min(m,n)) where k is the bandwidth, specified in the project description as 3 * 2 + 1 = 7, m is the depth of the left sequence, and n is the depth of the top sequence. Take n to be min(m,n) for the sake of this writeup.
        **i.** **[1]**(k * max(m, n)) + **[2]** (n*k + n*k) + **[3]** (m+n)
            1. The first item is the creation of the K*N matrix. The runtime here is going to be roughly K*N.
            2. This is the result of only filling in the values of the matrix contained within the band. The loop basically runs 2k either m or n times, depending on which is shorter. The loop ends after either m or n is reached. So, the main body runs in about **2O(n*k)** time.
            3. The third is the absolute worst case scenario that the return path could take. In this implementation, it would never get close to that because the path back is contained within the band traveling diagonally across the matrix. In reality, the average complexity for traversing all of the backtraces in this implementation is roughly n, where n is the depth of the array.
        ii. The full runtime simplifies to about **2O(kn)**. The line-by-line breakdown of the runtime can be found in the method body of the submitted code - **banded_needleman_kn().**
    b. Memory Complexity - This implementation uses **O(k*n)** memory.

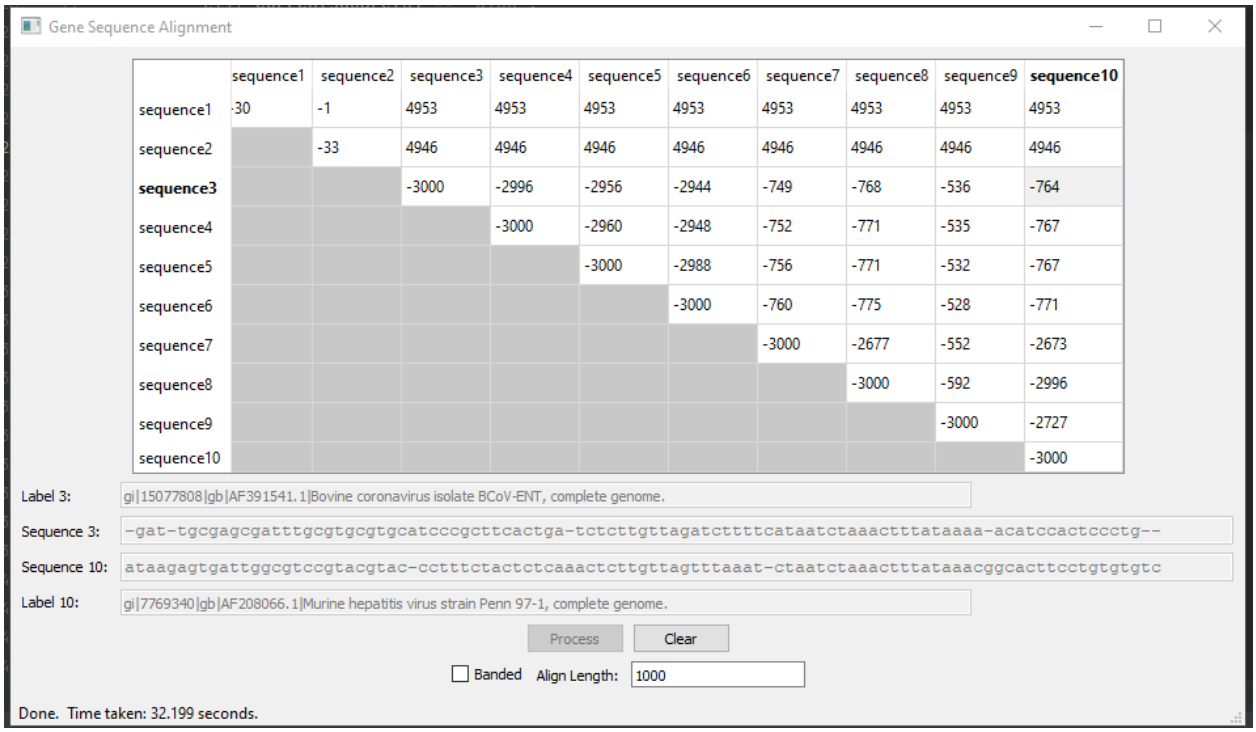
3. Banded Needleman-Wunsch **(Full Matrix - Functional)**
    a. Runtime Complexity - O(k*min(m,n)) where k is the bandwidth, specified in the project description as 3 * 2 + 1 = 7, m is the depth of the left sequence, and n is the depth of the top sequence.
        i. **[1]**(1 + k + k) + **[2]** (min(m,n)*k + min(m,n)*k) + **[3]** (m+n)
            1. The first item is the result of a bit of time saving I made to make the algorithm run in (k*m) time. It does, however, come at the sacrifice of memory. I create a matrix of size [align length][align length] before the main driver loop and copy it to the method every time a new sequence pair is called. Then, the base case is filled in for both directions, which takes 2k time. Instead of creating (i*j) matrices of different sizes, one matrix large enough to accommodate every problem in (i*j) is created and then copied.
            2. This is the result of only filling in the values of the matrix contained within the band. The loop basically runs 2k either m or n times, depending on which is shorter. The loop ends after either m or n is reached. So, the main body runs in about **2O(n*k)** time.
            3. The third is the absolute worst case scenario that the return path could take. In this implementation, it would never get close to that because the path back is contained within the band traveling diagonally across the matrix. In reality, the average complexity for traversing all of the backtraces is going to be close to **sqrt(m^2+n^2)**.
        ii. The full runtime simplifies to about **2O(k(min(m,n))**. The line-by-line breakdown of the runtime can be found in the method body of the submitted code - **banded_needleman().**
    b. Memory Complexity - A constant **O(L^2)** memory is used, where L is the specified alignment length. This was done to make the regular banded Needleman-Wunsch run in the specified O(k*n) runtime.

**Alignment Strategy**

I used a tuple for my matrix elements. It is of the form (string direction, int score). We start at the low right hand corner, and we follow the directions given by tuple[0], performing the corresponding string mutation at each step, until we reach the source, in the upper left corner. A travel upwards inserts “-” in the upper string, a travel left inserts “-” into the left hand string, and a diagonal is checked to see whether it’s a match or a mismatch, then either substitutes the value or treats it as an alignment and moves along. Then, the two resulting alignments are trimmed to the first 100 characters and returned to the GUI, along with the best score. This is done for both **unbanded_needleman()** and **unbanded_needleman()**, but not for **banded_needleman_kn()**. For more details, see **Conclusion**.

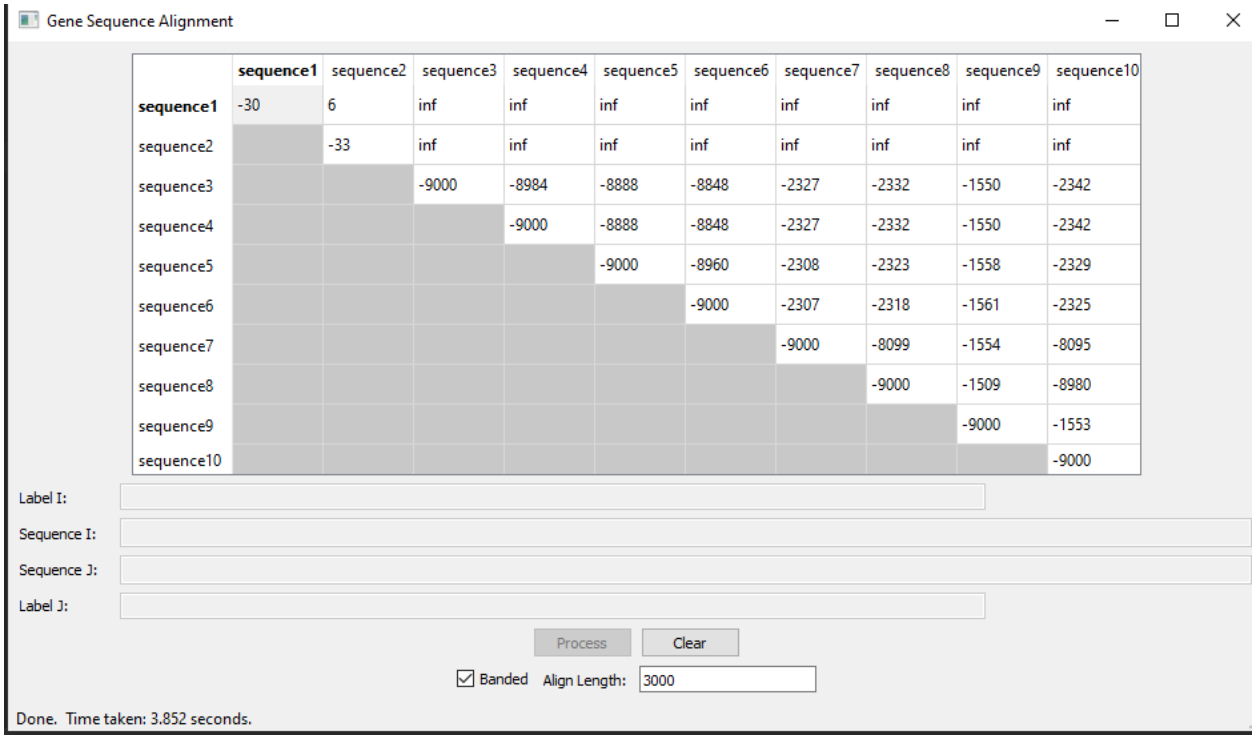
## Results

### 1. Unbanded - 1000




### 2. Banded, Full L*L Matrix

3. **Banded, K*N Matrix - See explanation below.**




| | sequence1 | sequence2 | sequence3 | sequence4 | sequence5 | sequence6 | sequence7 | sequence8 | sequence9 | sequence10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **sequence1** | -30 | 6 | inf | inf | inf | inf | inf | inf | inf | inf |
| sequence2 | | -33 | inf | inf | inf | inf | inf | inf | inf | inf |
| sequence3 | | | -9000 | -8984 | -8888 | -8848 | -2327 | -2332 | -1550 | -2342 |
| sequence4 | | | | -9000 | -8888 | -8848 | -2327 | -2332 | -1550 | -2342 |
| sequence5 | | | | | -9000 | -8960 | -2308 | -2323 | -1558 | -2329 |
| sequence6 | | | | | | -9000 | -2307 | -2318 | -1561 | -2325 |
| sequence7 | | | | | | | -9000 | -8099 | -1554 | -8095 |
| sequence8 | | | | | | | | -9000 | -1509 | -8980 |
| sequence9 | | | | | | | | | -9000 | -1553 |
| sequence10 | | | | | | | | | | -9000 |

## Conclusion

I was able to make the k*n implementation work, but only for comparisons of **equal length**. The more correct implementations are both my unbanded and full-matrix banded. The runtime and memory requirements are met by both the k*n implementation unbanded and the unbanded algorithms. Memory constraints were sacrificed for runtime for the full matrix implementation of Banded. The breakdown is as follows:

1. Unbanded - Performs within runtime and memory constraints, and is correct.
2. Banded - Full Matrix - Performs within runtime but **not** memory constraints, and is correct.
3. Banded - K*N Matrix - Performs within runtime and memory constraints, but is only **sometimes correct**.