Justin Mancinelli
42094353

Comp7505 Assignment 2

Problem 2

With regard to my implementation of the "branch-and-bound" variation of `findMedianKmer`, the worst-case running time depends on two variables. With respect to the number of sequences, it is exponential where the exponent corresponds to the number of sequences (see Appendix 1.1). With respect to the length of the k-mer, it is quadratic (see Appendix 1.2).

Appendix 1.3 contains a chart illustrating the practical running times for 2-25 sequences when k ranges between 5 and 12. There is some missing data because the benchmark program would skip subsequent iterations if the previous took longer than 30 seconds.

It is possible to reach the conclusion of quadratic running time with respect to the length of the k-mer by analyzing the source code. The implementation gradually increases the size of the prefix until size k but for analysis we can assume that the size begins at k and gradually decreases until 0. This gives us the recurrence relation $T(k) = O(S*N*k) + E*T(k-1)$ where S is the number of DNA sequences, N is the length of each sequence, E is the length of the alphabet and k is the length of the k-mer (see Appendix 1.4). To simplify the analysis, assume that S, N, and E are held constant (in practice, E and S are initialized before running the algorithm so this assumption has merit, however, it is possible to have several values of N but we do have control over this before the algorithm is run). With the stated assumptions, the recurrence relation simplifies to $T(k) = O(k) + T(k-1)$ which has the solution $T(k) = O(k^2)$.

Justin Mancinelli
42094353

Problem 3

The method responsible for updating the trie is `putKmer` which has a theoretical running time of O(E*k) where k is the length of the k-mer and E is the length of its alphabet (see Appendix 2.1). Since the k-mer's alphabet is constant, O(E*k) = O(k) which means `putKmer` is linear with respect to the length of the k-mer.

The method which accesses the counts from the completed trie is `getCount` which has a theoretical running time of O(k) (see Appendix 2.2). It is possible that we reach the end of the path before finding the k-mer but the worst-case is actually finding the k-mer since that will lead to a longer path.

Problem 4

The practical running time of `findMedianKmer` after improvements have been made is much faster (see Appendix 4.1). Using the trie to choose an optimal branching order helps minimally compared to reusing distances in this case. The reason is that the alphabet we're using is small so choosing a sub-optimal branch has little impact however recalculating distances for every recursive call becomes extremely expensive as the number of sequences and length of the k-mer grows.

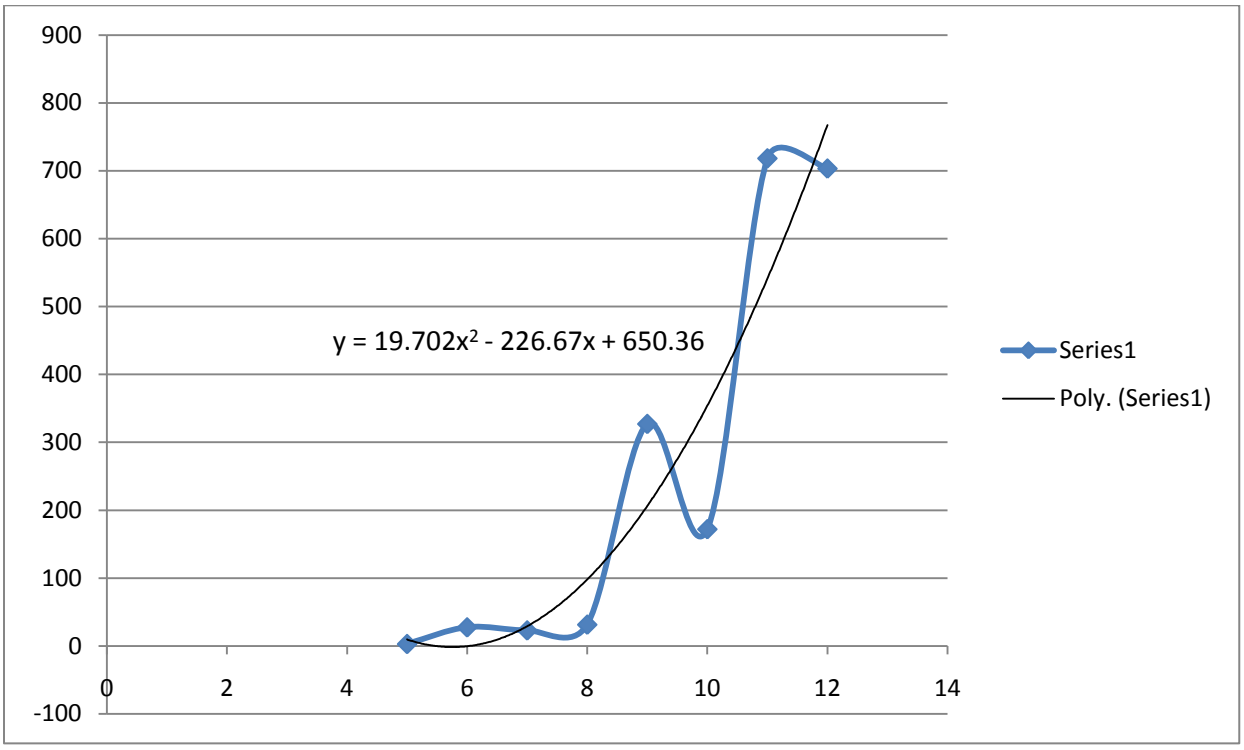
Specifically, `getDistance` and `getDistances` are O(n*k) where n is the length of the sequence and k is the length of the k-mer. The result must be evaluated over all sequences, so, in practice every distance calculation runs in O(n*k*N) where N is the number of sequences. By reusing distance calculations, in my implementation, we only use `getDistances` in the above way once. Every other call to `getDistances` thereafter happens within `getExtDists` which always only calculates the distance for a k-mer of length 1 so the running time of `getDistances` within `getExtDists` is O(n*N). The method `getExtDists` then sums the new, suffix distance with the previously calculated prefix distance which is also O(n*N). Therefore, by reusing distances, we cut the running time of finding distances from O(n*k*N) to O(n*N) which is a dramatic improvement.

Justin Mancinelli
42094353

Appendix 1.1

Plots of running time (in milliseconds) vs. k-mer length where n is the number of sequences.

Figure 1 -- Plot for n = 1



Justin Mancinelli
42094353

Figure 2 -- Plot for n = 2

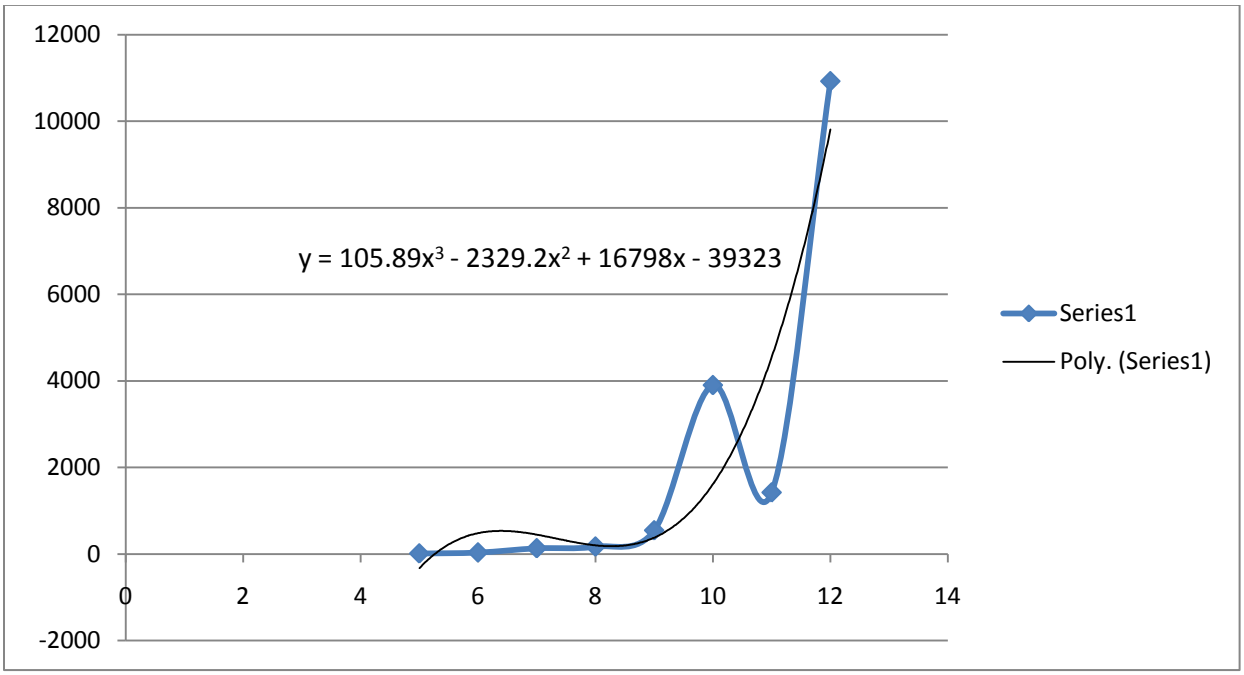
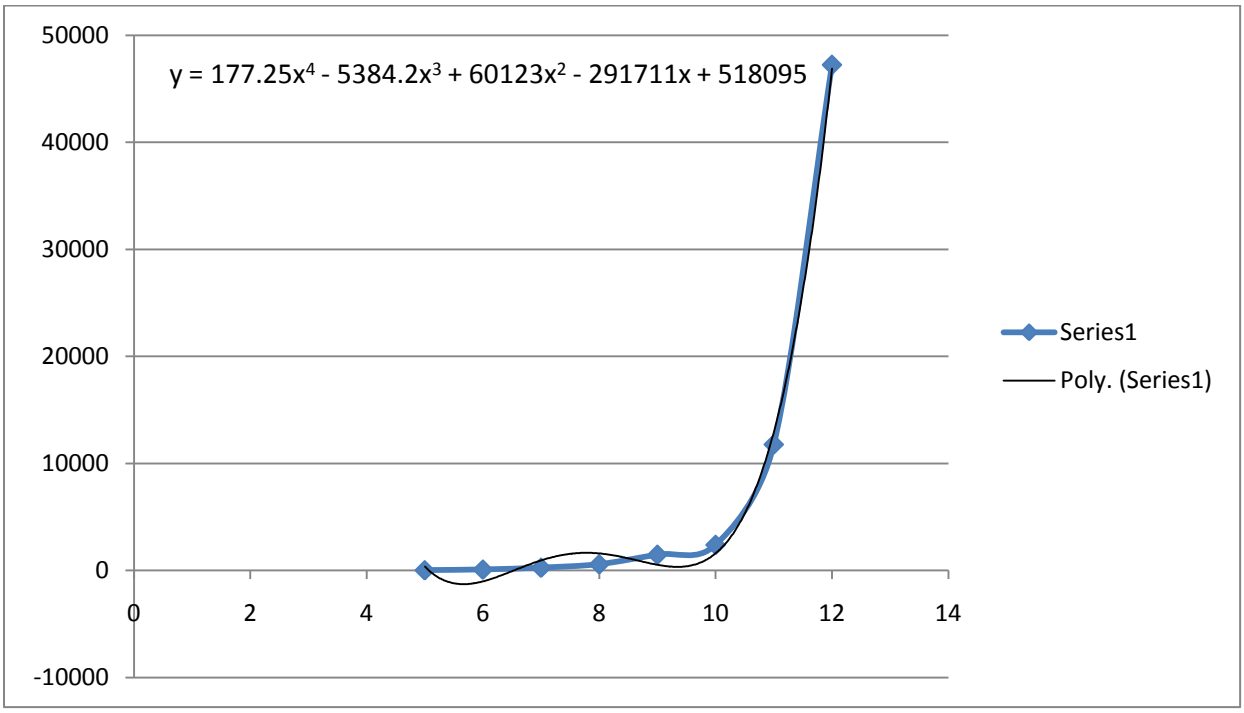


Figure 3 -- Plot for n = 4



Justin Mancinelli
42094353

Figure 4 -- Plot for n = 5

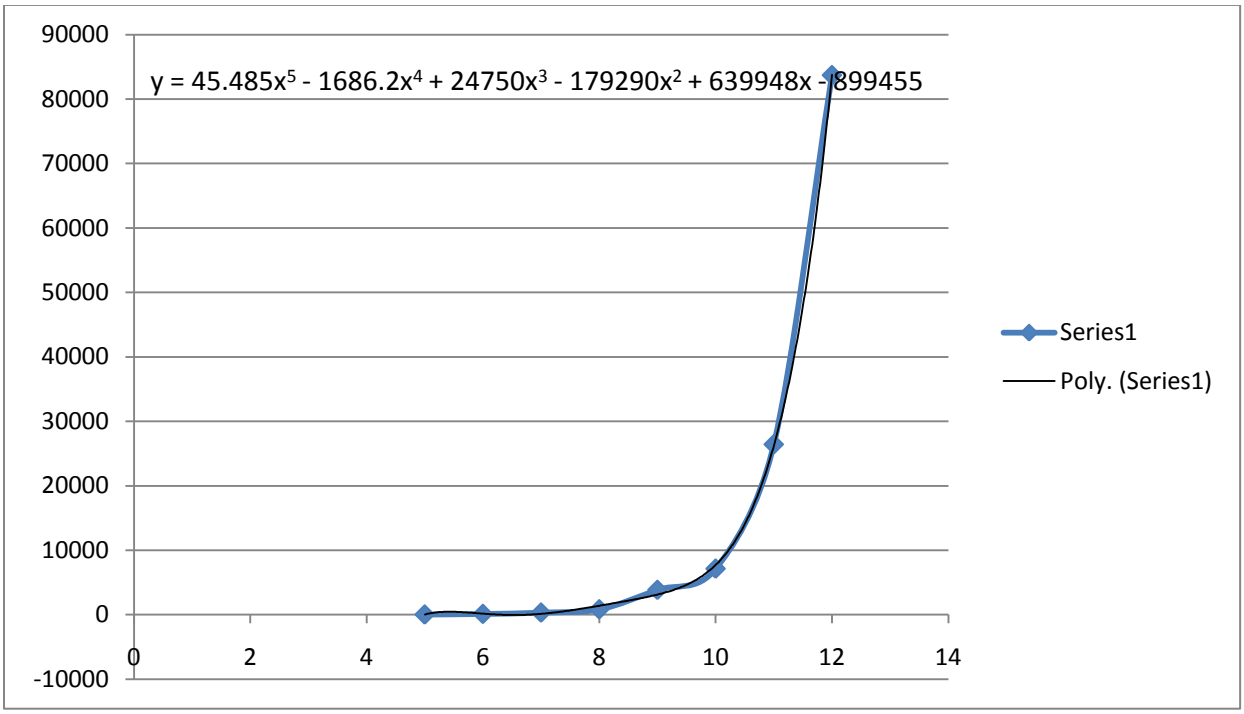
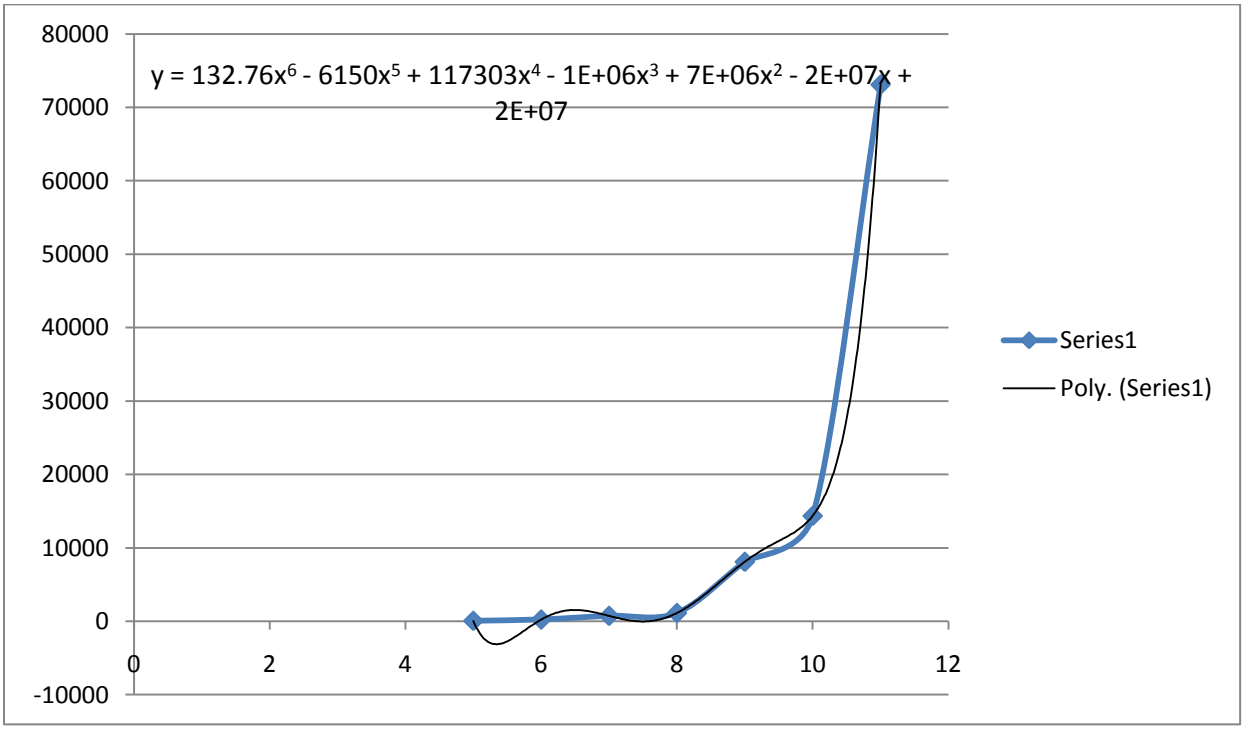


Figure 5 -- Plot for n = 6



Justin Mancinelli
42094353

Appendix 1.2

Plots of running time (in milliseconds) vs. number of sequences where k is the length of the k-mer.

Figure 6 -- Plot for k = 5

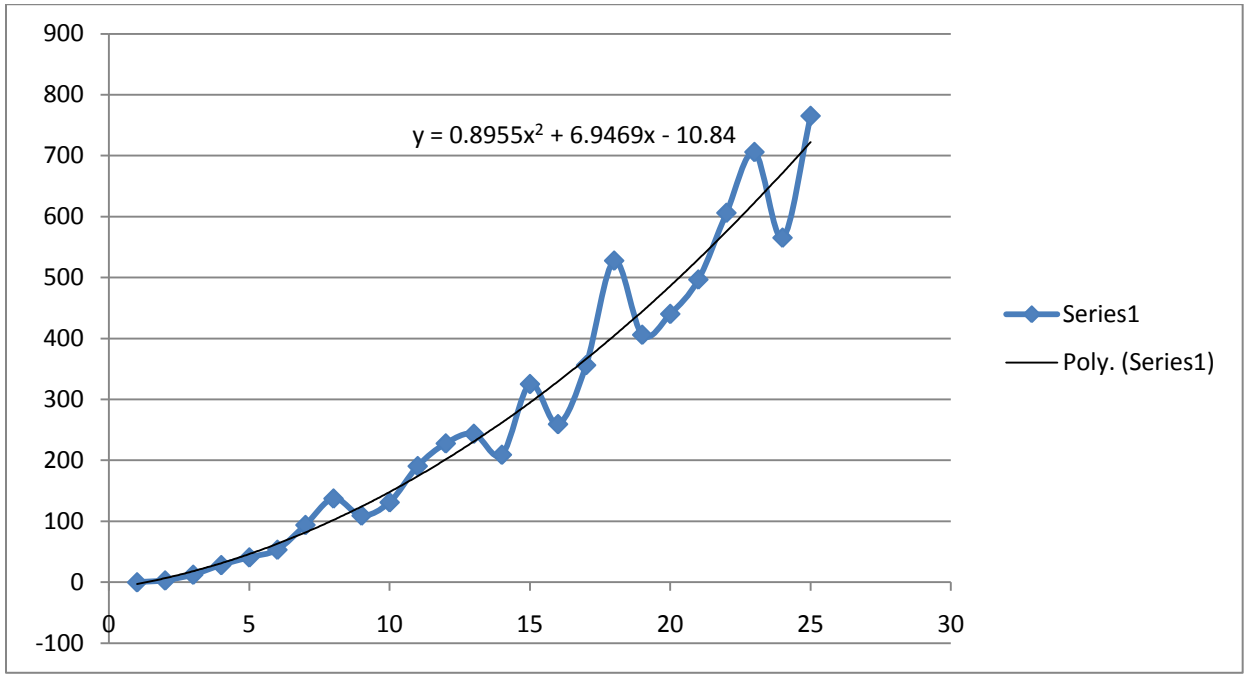
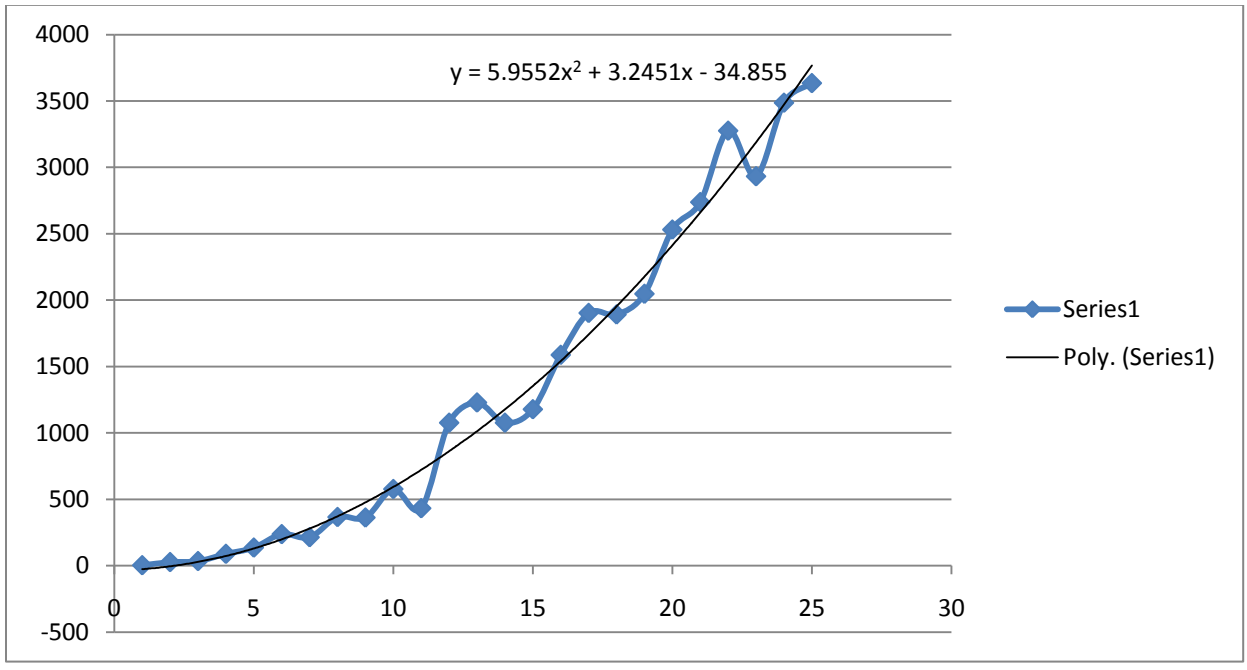


Figure 7 -- Plot for k = 6



Justin Mancinelli
42094353

Figure 8 -- Plot for k = 7

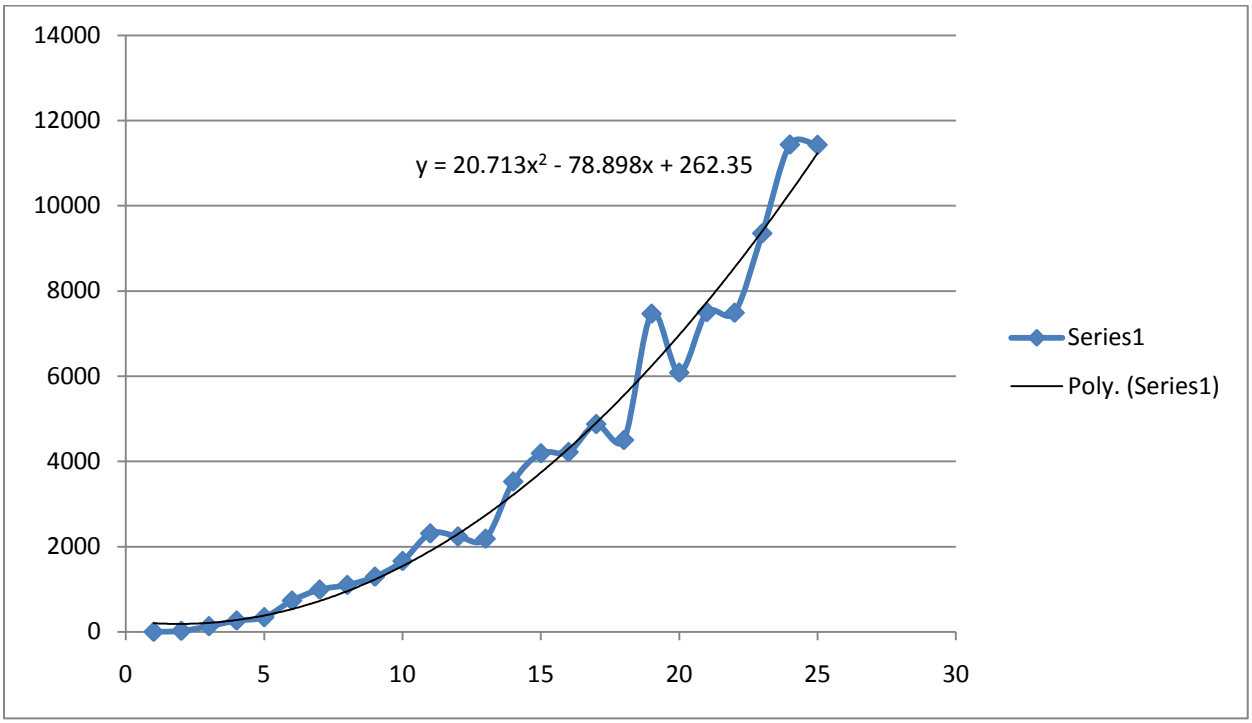
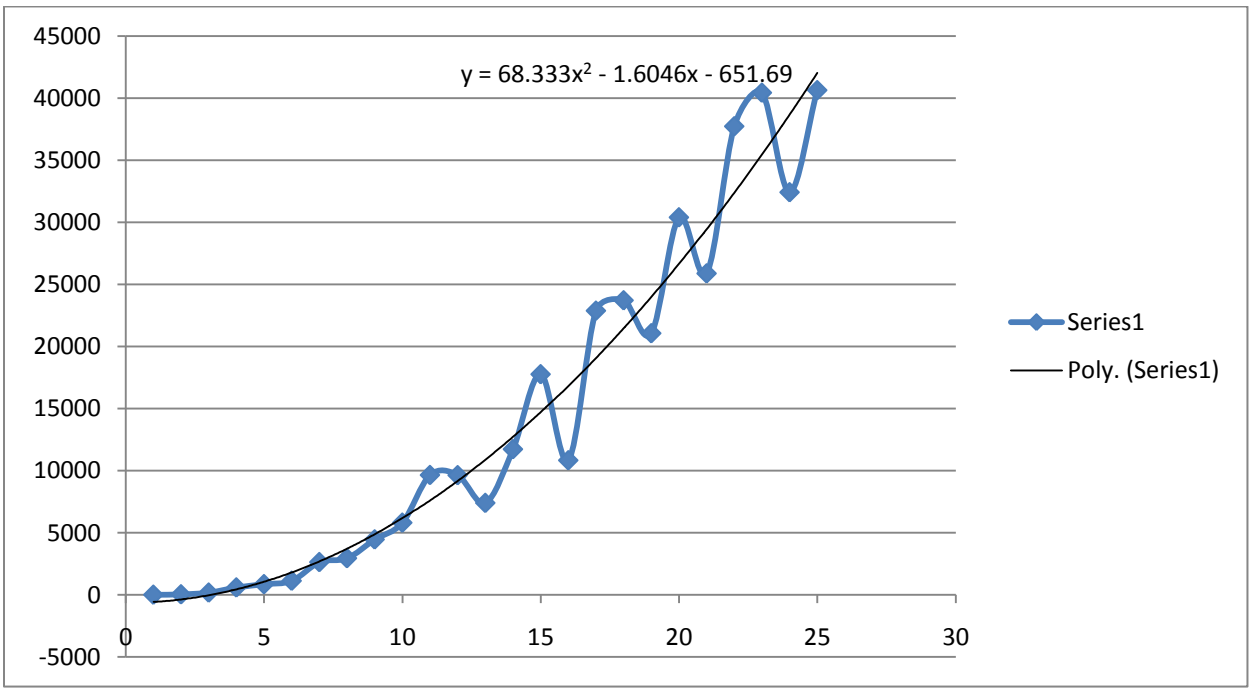


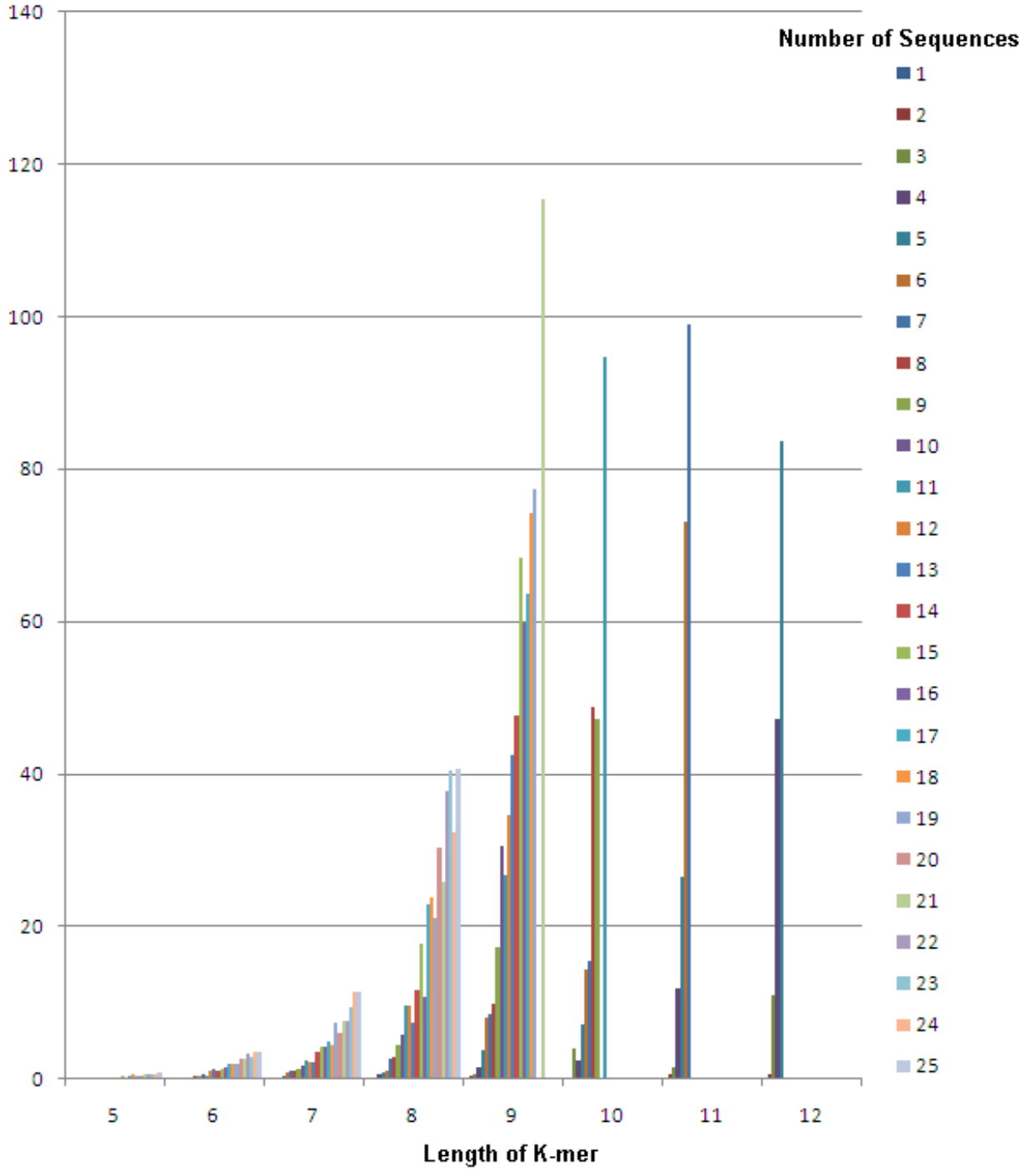
Figure 9 -- Plot for k = 8



Justin Mancinelli
42094353

Appendix 1.3

Runtime (in seconds) vs Length of K-mer and Number of Sequences for implementation of `findMedianKmer` which leverages branch-and-bound heuristic.



Justin Mancinelli
42094353

Appendix 1.4

Asymptotic analysis of `findMedianKmer`.

```
public Distance findMedianKMer(KMer prefix, int k, Distance old_best)          T(k) = O(S*N*k) + E*T(k-1)
    {
        if (prefix == null)                                                    O(1)
            prefix = new KMer(alpha, k);                                       O(k)

        if (prefix.isComplete()) {                                             O(1)
            int distance = 0;                                                  O(1)
            for (DNASequence seq : this.seqs) {                                O(S)*O(N*k) = O(S*N*k)
                distance += getDistance(seq, prefix);                          O(N*k) where N = |seq|, k = |prefix|
            }

            return new Distance(distance, prefix);                             O(1)
        }
        Distance best_dist = new Distance(k * this.seqs.length + 1, prefix);   O(1)

        if( old_best != null) {                                                O(1)
            int distance = 0;                                                  O(1)

            for (DNASequence sequence : this.seqs)                             O(S)*O(N*k) = O(S*N*k)
                distance += getDistance(sequence, prefix);                     O(N*k) where N = |seq|, k = |prefix|

            if (distance >= old_best.actual) {                                 O(1)
                return old_best;                                               O(1)
            }
        }
        Distance temp_dist;                                                    O(1)
        for (KMer kmer : prefix.getExtensions()) {                             O(E*?) Where E = |alpha|
            /* Keep old_best throughout traversal */
            if( old_best!=null) {                                              O(1)
                if (best_dist.actual >= old_best.actual)                       O(1)
                    best_dist = old_best;                                      O(1)
            }

            temp_dist = findMedianKMer(kmer, k, best_dist);                    ?
            if (temp_dist.actual <= best_dist.actual) {                        O(1)
                best_dist = temp_dist;                                         O(1)
            }
        }

        return best_dist;                                                      O(1)
```

Justin Mancinelli
42094353

Appendix 2.1

Asymptotic analysis of `putKmer`.

```
public void putKMer(TrieNode parent, KMer kmer) {                    
    int[] symbols = kmer.getKMer();                                  O(1)

    for (int i = 0; i < symbols.length; i++) {                       O(E*k) where k = |kmer|
        TrieNode current = parent.children[symbols[i]];              O(1)

        if (current == null) {                                       O(1)
            current = new TrieNode(this.alpha);                      O(E) where E = |alpha|
            current.symbol = symbols[i];                             O(1)
            parent.children[symbols[i]] = current;                   O(1)
        }

        current.count++;                                             O(1)

        parent = current;                                            O(1)
    }
}
```

Justin Mancinelli
42094353

Appendix 2.2

Asymptotic analysis of `getCount`.

```
public int getCount(KMer kmer) {                                        O(k)
    /* The path t0 follow */
    int[] path = kmer.getKMer();                                        O(1)

    /* The current node of the trie */
    TrieNode current = this.root;                                       O(1)

    /*
     * The count. If there is nothing in the tree then it's not possible to
     * find the kmer so default to 0
     */
    int count = 0;                                                      O(1)

    /* The depth which has been traveled */
    int depth = 0;                                                      O(1)

    /*
     * Continue traversing the path until either we have traversed the whole
     * path or we have reached a leaf node
     */
    while (depth < kmer.getLevel() && current.children[path[depth]] != null) {  O(k)
        /* go to the correct child and increase the depth */
        current = current.children[path[depth++]];                      O(1)
    }

    if (depth < kmer.getLevel()) {                                      O(1)
        /* Followed the path but didn't find kmer */
        return 0;                                                       O(1)
    } else {
        /* Found kmer ending at current node */
        count = current.count;                                          O(1)
    }

    return count;                                                       O(1)
}
```

Justin Mancinelli
42094353

Appendix 3.1

Runtime (in seconds) vs Length of K-mer and Number of Sequences for implementation of `findMedianKmer` which leverages the trie data structure and re-using distances of parent nodes.