Genome Rearrangement

Fernando Javier López Cerezo

December 9, 2021

1 Objectives

The objective of this project is to estimate the complexity of an algorithm which given a permutation, finds the shortest sequence of reversals required to transform it into the positive identity permutation. In particular we want to find the average complexity of an algorithm implementing a greedy algorithm approach and compare it to another algorithm implementing a more naive apporach.

2 Experimental Setup

For this project we used:

- MacBook Air 13 (1,8 GHz Dual-Core Intel Core i5 CPU, 8 GB 1600 MHz DDR3 RAM and macOS Catalina 10.15.5 OS)
- Java 15.0.2 and R 4.1.1

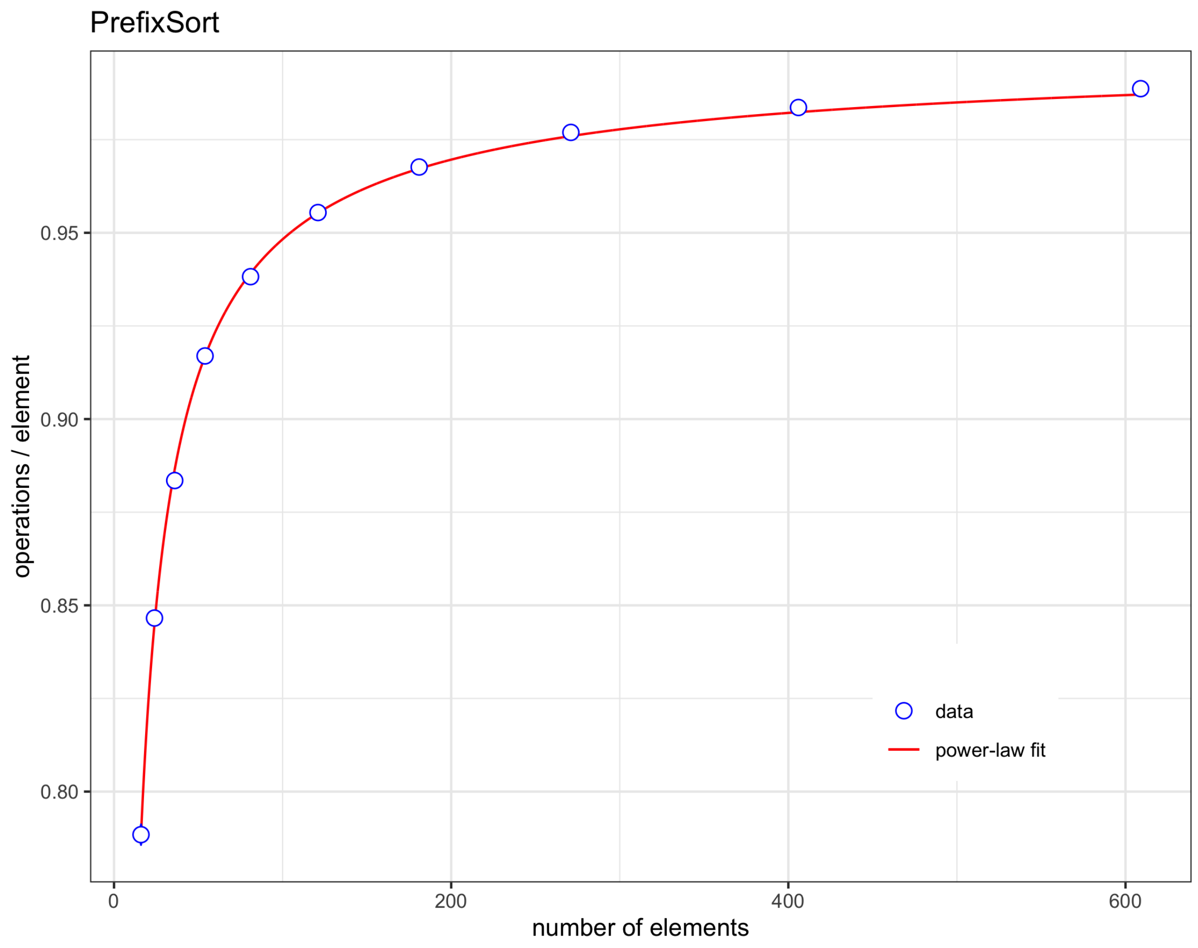
The method we used to obtain our data was the following:
We implemented an algorithm using a greedy approach using Java. The greedy idea behind our algorithm is picking the reversal that maximizes the number of breakpoints removed each time until we obtain the positive identity premutation. The algorithm we used to do this is very straightforward. Firstly we frame the permutation we receive as input and proceed to get all applicable operations (all the reversal candidates). We then pick the best operation by choosing the reversal with the best quality out of the candidates. We apply this operation to our permutation and get a list of new candidates. We repeat this procedure and once this list is empty we stop the process as we will have obtained the positive identity permutation. Finally we returned a list with all of the reversals we used along the way.

For the naive approach we just implemented an algorithm that detects the first position where our permutation is different from the identity, finds where in the permutation the desired value is and reverses the whole segment. This process is repeated over and over until we obtain the identity permutation.

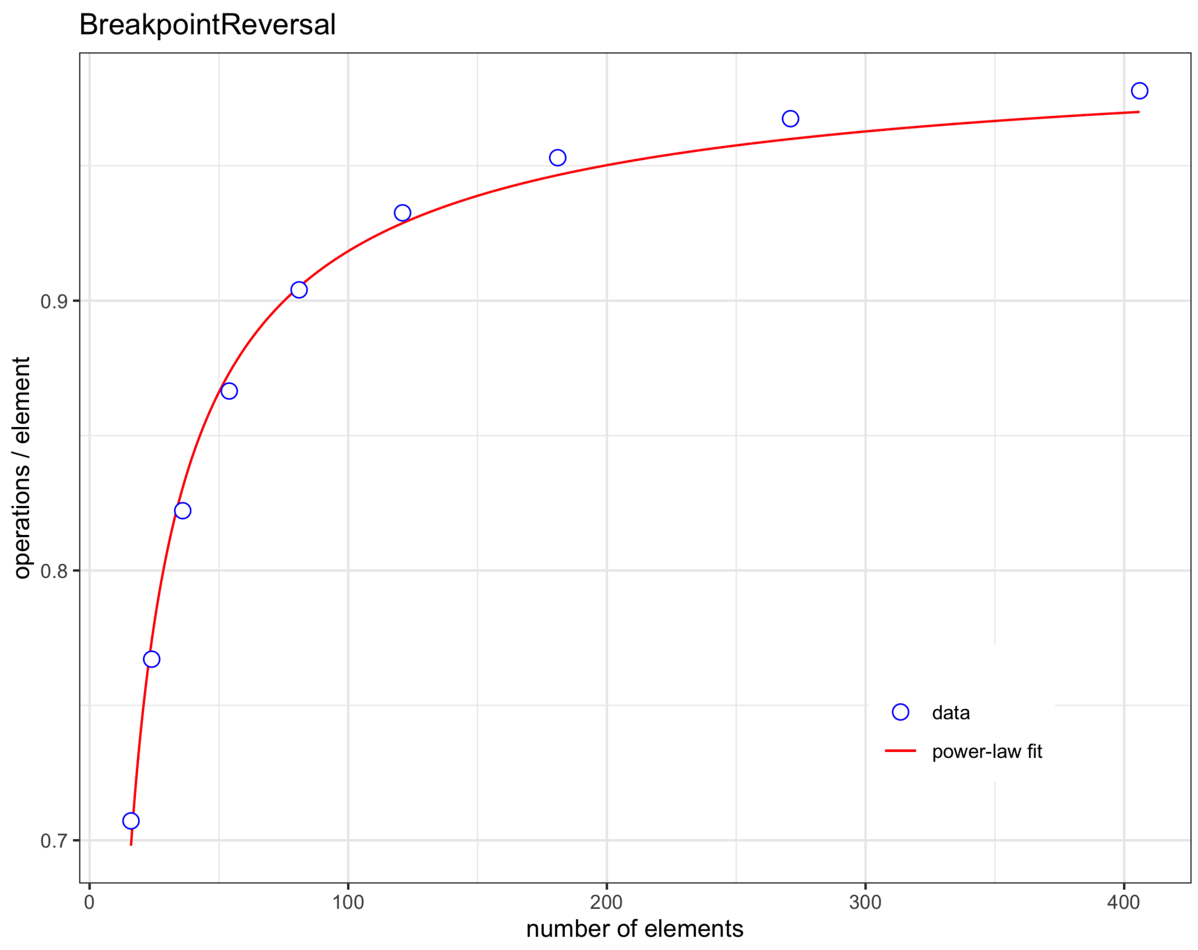
We used an algorithm to determine the time required to find the shortest sequence of reversals for both implementations. We used permutations with an initial length of 16 and a size increment of 10, each size was tested with 1000 permutations. At last all of this data was anaylzed using R and studied.

3 Empirical Results

Using R we plotted the following graph showing how the number of operations varies with the number of elements in the genome sequence.



R approximated our data to a power-law fit. Simply put, if we call our independent variable (number of elements) n, it plotted the function of the type $an^b \quad a, b \in \mathbb{R}, b > 1$ that best fitted our data. R also gave us estimated values for the graph of the type $an^b \quad a, b \in \mathbb{R}, b > 1$ that best fitted our data if we plotted it using the formula: 1 - number of operations. If we plotted a graph using this formula its estimation would be $1.786563n^{-0.769101}$



R approximated once again our data to a power-law fit. R also gave us estimated values for the graph of the type $an^b \quad a, b \in \mathbb{R}, b > 1$ that best fitted our data if we plotted it using the formula: 1 - number of operations. If we plotted a graph using this formula its estimation would be $2.18585n^{-0.71387}$

4 Discussion

We are interested in estimating the order of growth of the computational cost of our algorithm when the number of elements in the genome sequence becomes arbitrarily large. Lets call this order of growth $t(n)$ where n is the number of elements.

In this case it is clear by looking at our data that both our naive and greedy algorithms follow a power-law pattern of growth. In other words $t(n) = n^b \quad b \in \mathbb{R}, b > 1$ for both algorithms. In order to determine which of them is less efficient we should look at which one asymptotically tends to 1 the fastest when the number of elements become arbitrarily large. Looking at the graphs it seems like the greedy algorithm tends more slowly to 1 when the number of elements tend to infinity so it looks like it's the more efficient option. Looking at the data we obtained from R it is easy to see that when considering the graphs plotted when using the formula: 1 - number of operations the formula of the graph that has the lowest absolute value in the exponent will tend to 1 the slowest and therefore be the more efficient algorithm. Comparing the formula of both algorithms we can conclude that the greedy algorithm is more efficient than the naive approach to our problem.

Finally it should be taken into account that we studied the average complexity of this algorithms, normally we would be interested in the worst-case complexity. Also our greedy algorithm wasn't completely optimal, when choosing the best candidates if there was a tie in their quality one was chosen arbitrarily. If an adequate tie-breaking procedure is picked it would provide a 2-approximation to the optimal.