**Benchmarking Report**

**Note: for all methods, 50000 of its entries are randomly generated. Some of these entries exist within the set of the defined Morse codes and some don’t. However, we take the time taken to decode 50000 Morse codes to letters and repeat it 100 times. The average and median time is then taken to help us get an idea of their performance.**

**Method 1**

Data is stored in a json object as follows: [{letter: ‘a’, code: ‘1001’}, {}, ……]. To decode the given Morse code, we linearly iterate though each json objects until we find the correct code. Its corresponding letter is then printed out. If it the code does not exist, the program will have iterated through the full array and printed null.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Average time (ms) | 524.09 | 531.57 | 541.52 | 526.71 |
| Median time (ms) | 492 | 510 | 539.5 | 504.5 |

**Method 2**

Morse code data is stored as follows in a json object: : [{letter: ‘a’, code: [‘1’,’0’,’0’,’1’]}, {}, ……]. Decoding in this method is also similar to method 1 except we linearly iterate though a each element of the Morse code array to compare if they are equal. In method one, all we had to do was use the equality operator. If it the code does not exist, the program will have iterated through the full Morse code data array and printed null.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Average time (ms) | 525.87 | 534.29 | 529.51 | 544.14 |
| Median time (ms) | 497.5 | 513 | 505.5 | 548.5 |

**Method 3**

Morse codes are stored in an array as follows: ['1001', '0011', '0101', '1110', '1111'], where the first entry represents ‘a’, second is ‘b’, third is ‘c’ and so forth. When decoding a given entry, we use arrayCodes.indexOf(input) to get the corresponding index value. If the entry doesn’t exist, the index will be returned as -1. After getting the index, we use it to get the letter using the following line of code: String.fromCharCode(index+97). If it the code does not exist, the program will have iterated through the full Morse code data array and printed null.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Average time (ms) | 525.4 | 517.65 | 515.88 | 527.13 |
| Median time (ms) | 506.5 | 489.5 | 489 | 498 |

**Comparison**

As we can see, the differences in average times and median times are quite consistent with each other. This means that the test spikes are stable. By looking at the data however, the best performing one we could really pick out, based on the median times is Method 3 since it seems to have a much more stable median time which is slightly lower than the other two.

The median time is taken to measure performance since with the average case, a couple of results tend to have incorrect values. For instance, during one of the 100 times of testing the 50k morse code variations, a high priority process may have run by the OS, putting that particular test on hold and thus giving it an unreasonable time. This can affect the average whereas median won’t be.

Aside from performance, we also found it more straightforward to program using method 3. Since the morse codes are stored in an array, we can just use a built in function in node.js, array.indexOf(), to find the required letter. Also, it requires less memory since only the Morse codes have to be stored and not the corresponding letters. This is important since we are decoding on the client side, which means that we need to make the program as compact and efficient as we possibly can

On a side note, it is important to notice that when comparing two items for similarity, it always has a complexity of O(N). In general, all these methods find the matching Morse Code, O(N), and access the corresponding letter, O(1). This is why all these methods have very similar timings. That being said however, both methods (1 and 2) use json object to store data, whereas method 3 just uses a simple array. Json objects tend to be somewhat wasteful (although very useful in many applications), this is what gives Method 3 the advantage over the other two in terms of performance.

**Recommendation**

Method 3 will be our recommendation

It requires lesser lines of code and performs similar if not slightly better compared to methods 1 and 2.