

C-labben
Comparing Rabin-Karp with Naive Text
Search: An Analysis of String Matching
Algorithms

1 Summary

Text search algorithms have different strengths and weaknesses depending on the application scenario. In the case of searching for a pattern in a DNA sequence we observe different aspects of performance for Rabin-Karp and naive search algorithms. While both algorithms have the worst case time complexity of $O(n * m)$, where n is the length of the text and m is the length of the pattern, we observed that Rabin-Karp has an average-case time complexity of $O(n + m)$. However this difference deepens on the correct implementation of a hash function that is the integral part of the Rabin-Karp method.

Our Results clearly indicate this relation in time-complexity. When we doubled the length of the pattern the time for Rabin-Karp increased linearly from $21ms$ to $26ms$ while the time for naive search doubled. For naive search the time for n behaved as $O(n * m) \sim 228ms$ while the time for $2n$ jumped to $O((2 * n) * m) \sim 432ms$ on the other side the Rabin-Karp for n $O(n + m) \sim 21ms$ increased to $O(2n + m) \sim 26ms$ for case of $2n$.

While the naive search was easy to implement it outperformed Rabin-Karp only in niche cases with relatively small length of search pattern. On the other side while the Rabin-Karp algorithm can be more time efficient it is susceptible to hash collisions, which can result in false positives.

When it comes to scalability the Rabin-Karp outperforms the naive search by a noticeable margin however one has to note the memory required to store hash values. In certain scenarios this memory cost can be excessive especially when searching for multiple patterns simultaneously. We also have to note that for extremely long search patterns the cost of computing hash values might not be justified.

The choice between naive search and Rabin-Karp should take into account the trade-offs between time complexity, scalability, and false positives, as well as other factors such as ease of implementation and available computational resources.

2 Task description

In this laboratory, the primary objective is to compare two distinct string search algorithms based on two pertinent aspects in a particular scenario. More specifically, we will conduct a comparison between the Rabin-Karp algorithm and the naive string search algorithm in the context of searching for a pattern within a DNA sequence. We will focus on aspects of Time complexity, Scalability, Ease of implementation and Accuracy- false positives.

3 Method

I decided to compare the Rabin-Karp and the naive text search based on their performance in a scenario of searching for a pattern in a DNA sequence. The comparison is based on time needed to find a pattern in a DNA text file. The goal was to obtain times needed to find a pattern in a text file for different lengths of pattern and text.

More precisely for each length of pattern [5, 10, 25, 50, 100, 250, 500, 1000] we generated a random string of varying lengths [1000, 5000, 100000, 500000, 1000000, 2500000, 5000000] and measured how long each algorithm takes to find the pattern. Each measurement was repeated a 1000 times and the arithmetic mean taken as a result. Since the naive search method is relatively simple and there is not much one can change about it I also focused on implementing different hash functions in Rabin-Karp algorithm to see how they effect the performance. I also implemented code that tests for false positives. These test aspects are important because we have to investigate different lengths of pattern in different lengths of text. In the scenario of DNA search scientist often have to work with incredibly long files when searching parts of chromosomes responsible for a specific protein synthesis. In these cases using the correct text search algorithm to reduce computational costs becomes more and more important. This example also illustrates the importance of avoiding false positives, as well as the importance of scalability.

4 Result

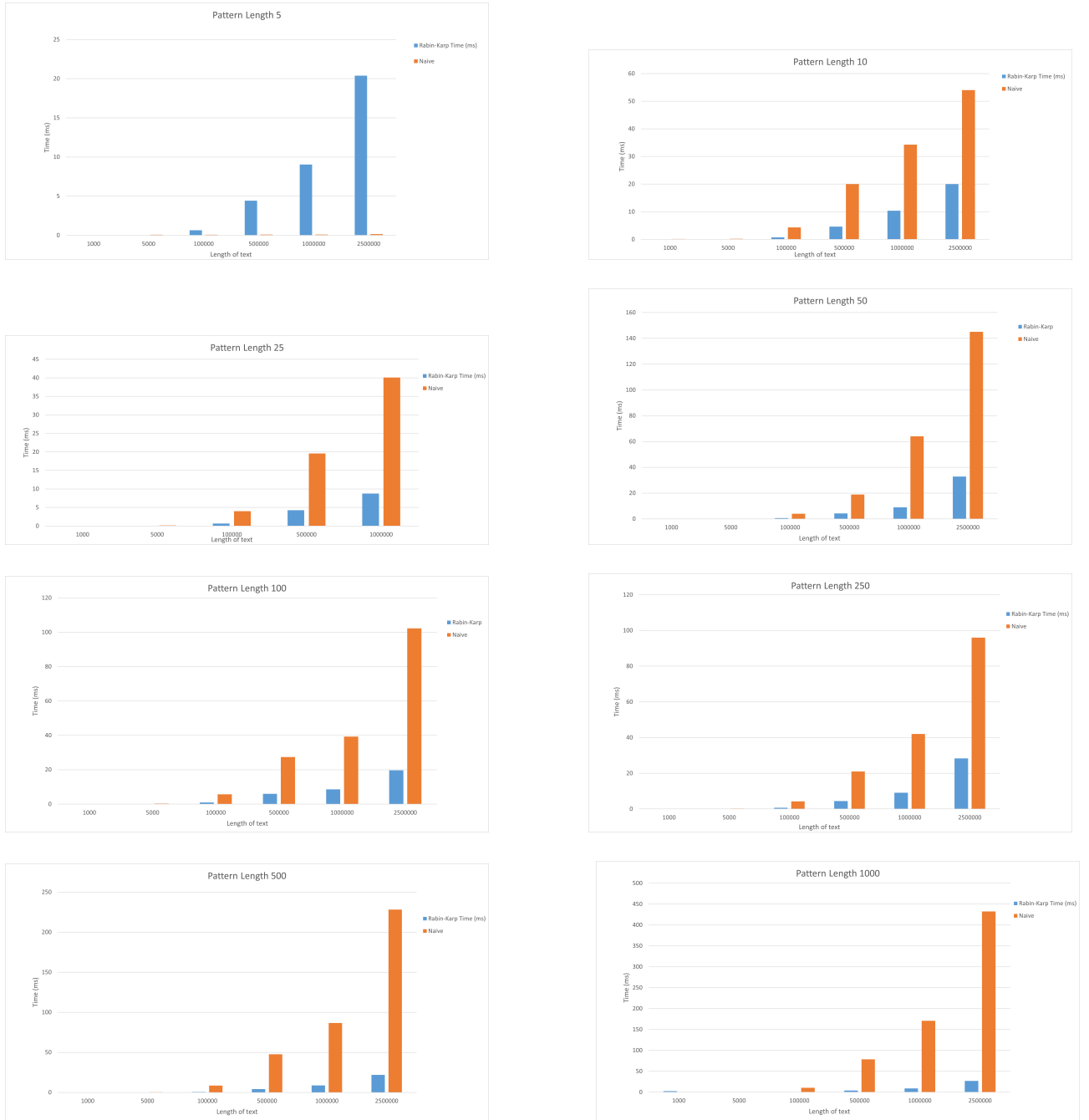


Figure 1: Average search times in $[ms]$ for patterns of varying lengths in text of different lengths measured in number of characters.

For more details please see appendix. There you can find exact times including false positives as well as results when testing different hash functions for Rabin-Karp.

5 Analysis ¹

The naive search algorithm has a time complexity of $O(n*m)$. The Rabin-Karp algorithm has a time complexity of $O(n * m)$ in the worst case, however if a good hash function is used, the average-case time complexity is $O(n + m)$.

The scenario of searching for a pattern in a DNA sequence was chosen for several reasons. DNA sequences are a great example of why choosing a correct text search algorithm can be extremely important. Since the input sizes for DNA sequences are often extremely long, often several millions of base pairs they are also a great example to illustrate the differences. It has to be noted that the length of the alphabet can have an impact on the performance of algorithms for DNA search. If the alphabet size were larger, it would require more space to store the hash values used in the Rabin-Karp algorithm, which could slow down the algorithm's performance. In this way the scenario was designed to favour Rabin-Karp algorithm.

Some straight facts of the naive search algorithm are that it is easy to implement, that it does not require additional memory to store hash values and that it works great for short search patterns. However since it compares every character of the pattern with every character of the text file it can be needlessly time-consuming.

On the other side the Rabin-Karp has better average time complexity of $O(n + m)$, if the optimal hash-function is used to compare the sub-strings. It can handle patterns with repeating characters more efficiently than some other string searching algorithms, as it uses a rolling hash function. However Rabin-Karp algorithm is susceptible to hash collisions resulting in false positives, it also requires additional memory to store the hash value's. Since the Rabin-Karp is heavily dependant on the hash function being used in some cases the cost of hashing part of the algorithm does not justify the time cost of doing it.

The hash function has to be carefully chosen to fit the problem scenario. For example if we look at the results for the built in python hash function and the Bernstein hash function in the appendix we clearly see that the naive search method outperforms the Rabin-Karp algorithm. To reach optimal performance the hash function has to be a rolling hash function. This means that unlike traditional hash functions that operate on the entire input string at once, the function processes the input string incrementally by updating the hash value as characters are added or removed. Even when the optimal hash-function ² is chosen for some cases the naive method performs better as we can see in the results for pattern length of 5 characters. The best way to describe this is via the allegory of the sliding window. Rabin-Karp works by sliding a window of size equal to the length of the pattern over the text and computing the hash value within this window. The Rabin-Karp avoids comparing each character of the text with the pattern by using a hash function to compare the sub-strings within the window, and it avoids recomputing the hash value of the entire sub-string every time the window moves by using a rolling hash function.

As we can clearly see from the results as the length of the text we are searching through increases the time required by the naive search algorithm starts increasing as expected $O(n*m)$ however the time required by the Rabin-Karp remains relatively unaffected. If we observe the case of searching for a pattern of length 500 in a 2,5 million nucleotides long sequence with searching for a pattern of length 1000 in a 2,5 million nucleotides long sequence, we observe that the time required by naive search doubled while the time for Rabin-Karp increased linearly from 21ms to 26ms demonstrating the naive search behaved according to $O(n*m) \sim 228$ ms to $O((2*n)*m) \sim 432$ ms with the case for Rabin-Karp $O(n+m) \sim 21$ ms to $O(2n+m) \sim 26$ ms.

When it comes to scalability Rabin-Karp algorithm should be used for small to medium lengths of search pattern. As i demonstrated it performs terrifically in the range 25 up to few thousand characters. Larger window increases the computational cost of computing the hash value for the pattern and the sub-strings. However when compared to naive search method the Rabin-Karp is more scalable.

It is important to note that the Rabin-Karp algorithm is a probabilistic algorithm, and there is always a chance of false positives even with a well-designed hash function. If the hash functions for the element within the observed window results in a clash the algorithm will produce a false positive. For example if the file being searched contains a lot of repeating patterns, the probability of hash collisions may increase. In our scenario of searching within a DNA sequence with a small alphabet as the pattern sequence increases in length we can passably get more false positives since more possible combinations of nucleotides that can produce the same hash value.

¹https://ocw.mit.edu/ans7870/6/6.047/f15/MIT6_047F15_Compiled.pdf

²<https://stackoverflow.com/questions/12324725/what-does-it-mean-for-a-hash-function-to-be-incremental>

6 Appendix

6.1 Using rolling hash function ³

```
import time
import random

def rabin_karp(text, pattern):
    n = len(pattern)
    m = len(text)
    q = 101 # A large prime number
    d = 4 # Size of the DNA alphabet (ACGT)
    h = pow(d, m-1) % q

    if m > n:
        return None

    # Convert the DNA sequences to arrays of integers using a mapping of ACGT to 0-3
    dna_map = {'A': 0, 'C': 1, 'G': 2, 'T': 3}
    p = [dna_map[c] for c in pattern]
    t = [dna_map[c] for c in text[:m]]

    # Calculate the hash value of the pattern and the first substring of the text
    p_hash = sum([p[i] * pow(d, m-1-i) for i in range(m)]) % q
    t_hash = sum([t[i] * pow(d, m-1-i) for i in range(m)]) % q

    # Matching
    for s in range(n - m + 1):
        # Check if the hash values match
        if p_hash == t_hash:
            # Check if the pattern matches the substring
            if pattern == text[s:s+m]:
                return s
        # Calculate the hash value of the next substring using rolling hash
        if s < n - m:
            t_hash = (d * (t_hash - t[s] * h) + t[(s+m)%n]) % q
            t[s%m] = dna_map[text[(s+m)%n]]

    return None

def naive_search(text, pattern):
    n, m = len(text), len(pattern)

    for s in range(n - m + 1):
        if text[s:s + m] == pattern:
            return s

    return None

def generate_dna_string(length, pattern):
    dna_string = ''.join(random.choice('ACGT') for _ in range(length))
    insertion_index = random.randint(0, length - len(pattern))
    dna_string = dna_string[:insertion_index] + pattern + dna_string[insertion_index + len(pattern):]
    return dna_string
```

³<https://www.delftstack.com/howto/python/rabin-karp-algorithm-in-python/>

```

def measure_execution_time(func,element,length,repitions=1000):
    total_time = 0
    for i in range(repitions):
        pattern = generate_random_pattern(element)
        text = generate_dna_string(length, pattern)
        start_time = time.time()
        func(text, pattern)
        end_time = time.time()
        total_time += (end_time - start_time) * 1000 # time in milliseconds
    return total_time / repitions

def generate_random_pattern(length):
    return ''.join(random.choice('ACGT') for _ in range(length))

def main():
    # DNA sequences of up to several million nucleotides in length
    text_lengths = [1000,5000,100000,500000,1000000,2500000]
    pattern_lengths=[50,100,250,500,1000]

    print("{:<15} {:<15} {:<25} {:<25}".format("Text Length", "Pattern Length",
                                             "Rabin-Karp Time (ms)", "Naive Search Time (ms)"))

    for element in pattern_lengths:
        for length in text_lengths:

            rabin_karp_time = measure_execution_time(rabin_karp,element,length)
            naive_search_time = measure_execution_time(naive_search,element,length)

            print("{:<15} {:<15} {:<25} {:<25}".format(length, element,
                                                    rabin_karp_time, naive_search_time))

if __name__ == "__main__":
    main()

```

6.2 Testing for false positives

```

def generate_dna_string(length):
    dna_string = ''.join(random.choice('ACGT') for _ in range(length))
    return dna_string

def generate_random_pattern(length):
    return ''.join(random.choice('ACGT') for _ in range(length))

def measure_false(func1,func2,element,length,repitions=1000):
    text = generate_dna_string(length)
    fpositive=0
    for i in range(repitions):
        pattern = generate_random_pattern(element)
        if func1(text, pattern) == True and func2(text, pattern)==False:
            fpositive+=fpositive
    return fpositive

```

| Text Length | Pattern Length | Rabin-Karp Time (ms) | Naive Search Time (ms) | False Positive |
|-------------|----------------|-----------------------|---------------------------|----------------|
| 1000 | 5 | 0.0 | 0.0306193828582763675000 | 0 |
| 5000 | 5 | 0.025917768478393555 | 0.07520198822021484100000 | 0 |
| 100000 | 5 | 0.6519463062286377 | 0.07499456405639648 | 0 |
| 500000 | 5 | 4.428615570068359 | 0.09029912948608398 | 0 |
| 1000000 | 5 | 9.031389474868774 | 0.09527945518493652 | 0 |
| 2500000 | 5 | 20.386231899261475 | 0.1521918773651123 | 0 |
| 1000 | 10 | 0.015018224716186523 | 0.11884069442749023 | 0 |
| 5000 | 10 | 0.03571295738220215 | 0.22313761711120605 | 0 |
| 100000 | 10 | 0.7362523078918457 | 4.31914210319519 | 0 |
| 500000 | 10 | 4.630545139312744 | 20.01859974861145 | 0 |
| 1000000 | 10 | 10.335901021957397 | 34.29586386680603 | 0 |
| 2500000 | 10 | 19.970502614974976 | 54.03413939476013 | 0 |
| 1000 | 25 | 0.002992391586303711 | 0.02708125114440918 | 0 |
| 5000 | 25 | 0.01731133460998535 | 0.16130518913269043 | 0 |
| 100000 | 25 | 0.657996416091919 | 3.9939064979553223 | 0 |
| 500000 | 25 | 4.241345643997192 | 19.594831466674805 | 0 |
| 1000000 | 25 | 8.766033172607422 | 40.11118793487549 | 0 |
| 1000 | 50 | 0.0019028186798095703 | 0.032912492752075195 | 0 |
| 5000 | 50 | 0.02191758155822754 | 0.1888260841369629 | 0 |
| 100000 | 50 | 0.6822161674499512 | 3.949831008911133 | 0 |
| 500000 | 50 | 4.188653230667114 | 18.901549100875854 | 0 |
| 1000000 | 50 | 8.897652387619019 | 64.06584930419922 | 0 |
| 2500000 | 50 | 32.82390522956848 | 145.02112793922424 | 0 |
| 1000 | 100 | 0.0069429874420166016 | 0.03898024559020996 | 0 |
| 5000 | 100 | 0.04350876808166504 | 0.2811610698699951 | 0 |
| 100000 | 100 | 0.9313819408416748 | 5.676714897155762 | 0 |
| 500000 | 100 | 5.998861789703369 | 27.40898895263672 | 0 |
| 1000000 | 100 | 8.625906229019165 | 39.27177381515503 | 0 |
| 2500000 | 100 | 19.671029567718506 | 102.26274061203003 | 0 |
| 1000 | 250 | 0.001794576644897461 | 0.03440999984741211 | 0 |
| 5000 | 250 | 0.028687477111816406 | 0.22872161865234375 | 0 |
| 100000 | 250 | 0.6991372108459473 | 4.282202243804932 | 0 |
| 500000 | 250 | 4.428828001022339 | 20.962100982666016 | 0 |
| 1000000 | 250 | 9.063412189483643 | 42.05052995681763 | 0 |
| 2500000 | 250 | 28.383312940597534 | 95.96701860427856 | 0 |
| 1000 | 500 | 0.0009963512420654297 | 0.05314064025878906 | 0 |
| 5000 | 500 | 0.023023605346679688 | 0.37652039527893066 | 0 |
| 100000 | 500 | 0.6285240650177002 | 8.473663091659546 | 0 |
| 500000 | 500 | 4.233642578125 | 47.7301709651947 | 0 |
| 1000000 | 500 | 8.689618587493896 | 86.60439944267273 | 0 |
| 2500000 | 500 | 21.973209381103516 | 228.23783206939697 | 0 |
| 1000 | 1000 | 2.3204755783081055 | 0.0009980201721191406 | 0 |
| 5000 | 1000 | 0.031098604202270508 | 0.42168378829956055 | 0 |
| 100000 | 1000 | 0.6751086711883545 | 10.627511739730835 | 0 |
| 500000 | 1000 | 4.146228075027466 | 78.44162201881409 | 0 |
| 1000000 | 1000 | 8.83913540840149 | 170.89659714698792 | 0 |
| 2500000 | 1000 | 26.792144298553467 | 432.33301305770874 | 0 |

6.3 Using built in python hash function

```
def rabin_karp(text, pattern):
    n, m = len(text), len(pattern)
    p_hash = hash(pattern)
    for i in range(n - m + 1):
        if hash(text[i:i + m]) == p_hash:
            if text[i:i + m] == pattern:
                #print("Found!")
            return i
    return None
```

| Text Length | Pattern Length | Rabin-Karp-Time (ms) | Search Time (ms) | False Positive |
|-------------|----------------|----------------------|---------------------|----------------|
| 1000000 | 5 | 0.0 | 0.20673274993896484 | 0 |
| 5000000 | 5 | 0.4981040954589844 | 0.0 | 0 |
| 1000000 | 10 | 69.77112293243408 | 48.256611824035645 | 0 |
| 5000000 | 10 | 199.0140676498413 | 51.5042781829834 | 0 |
| 1000000 | 20 | 234.92329120635986 | 36.18636131286621 | 0 |
| 5000000 | 20 | 949.9693632125854 | 246.72837257385254 | 1 |
| 1000000 | 50 | 219.71125602722168 | 38.02487850189209 | 0 |
| 5000000 | 50 | 1121.1829900741577 | 224.12827014923096 | 0 |
| 1000000 | 100 | 130.06658554077148 | 46.24142646789551 | 0 |
| 5000000 | 100 | 1164.3502712249756 | 290.78049659729004 | 0 |
| 1000000 | 250 | 273.8816261291504 | 37.18128204345703 | 0 |
| 5000000 | 250 | 1389.8430347442627 | 130.9541940689087 | 0 |
| 1000000 | 500 | 202.11443901062012 | 139.84980583190918 | 0 |
| 5000000 | 500 | 1063.6194705963135 | 355.2915573120117 | 0 |
| 1000000 | 1000 | 222.78985977172852 | 258.55042934417725 | 0 |
| 5000000 | 1000 | 898.6821889877319 | 396.80497646331787 | 0 |

6.4 Using modified Bernstein hash function

```
def rabin_karp(text, pattern):
    n, m = len(text), len(pattern)
    p_hash = bernstein_hash(pattern)
    for i in range(n - m + 1):
        if bernstein_hash(text[i:i + m]) == p_hash:
            if text[i:i + m] == pattern:
                #print("Found!")
                return i
    return None

def bernstein_hash(seq, d=33):
    h = 0
    for c in seq:
        h = (h * d) + ord(c)
    return h
```

| Text Length | Pattern Length | Rabin-Karp Time (ms) | Naive Search Time (ms) | False Positive |
|-------------|----------------|----------------------|------------------------|----------------|
| 1000000 | 10 | 71.05302810668945 | 34.908199310302734 | 1 |
| 2500000 | 10 | 139.48626518249512 | 65.87064266204834 | 2 |
| 5000000 | 10 | 106.30612373352051 | 57.56981372833252 | 0 |
| 1000000 | 20 | 208.89708995819092 | 51.068854331970215 | 1 |
| 2500000 | 20 | 519.6056127548218 | 131.03938102722168 | 0 |
| 5000000 | 20 | 801.1648416519165 | 210.74838638305664 | 2 |
| 1000000 | 50 | 184.79490280151367 | 45.01020908355713 | 0 |
| 2500000 | 50 | 357.8047513961792 | 64.38195705413818 | 0 |
| 5000000 | 50 | 925.163197517395 | 208.75282287597656 | 0 |
| 1000000 | 100 | 193.4767484664917 | 44.49634552001953 | 0 |
| 2500000 | 100 | 266.8581485748291 | 68.60189437866211 | 0 |
| 5000000 | 100 | 997.431468963623 | 189.64464664459229 | 0 |
| 1000000 | 250 | 162.94381618499756 | 26.816797256469727 | 0 |
| 2500000 | 250 | 543.7373876571655 | 109.12630558013916 | 0 |
| 5000000 | 250 | 944.2512273788452 | 152.394700050354 | 0 |
| 1000000 | 500 | 242.08426475524902 | 196.6578483581543 | 0 |
| 2500000 | 500 | 433.5677146911621 | 286.15753650665283 | 0 |
| 5000000 | 500 | 844.176983833313 | 481.4058303833008 | 0 |
| 1000000 | 1000 | 243.71204376220703 | 247.30727672576904 | 0 |
| 2500000 | 1000 | 516.1868095397949 | 195.32945156097412 | 0 |
| 5000000 | 1000 | 1317.4334049224854 | 417.6394462585449 | 0 |
| 1000000 | 10000 | 237.63983249664307 | 158.26287269592285 | 0 |
| 2500000 | 10000 | 484.772253036499 | 329.1821479797363 | 0 |
| 5000000 | 10000 | 893.8636302947998 | 739.9677038192749 | 0 |