String Matching Algorithms

Naive and KMP algorithms implementation and analysis

Elia Innocenti

November 2023

Index

1 Introduction

In this report we would like to compare two algorithms for the **string-matching problem**, namely the **naive string-matching algorithm** and the **Knuth-Morris-Pratt algorithm**.

1.1 Hardware and software specifications

Below are the specifications of the machine used to conduct the tests:

- **CPU:** Apple M1 SoC (System on Chip) with an 8-core CPU (4 high-performance and 4 high-efficiency cores), 7-core GPU and Neural Engine
- **RAM:** 8GB unified memory
- **OS:** Sonoma 14.0
- **Python version:** 3.11

1.2 References

Chapter 32 of the book *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (Third Edition).

2 The string matching problem

2.1 Definition

The **string-matching problem** is the problem of finding all occurrences of a pattern P in a text T.

We formalize the string-matching problem as follows.
We assume that the test is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of T and P are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, c, \ldots, z\}$, the set of lowercase English letters. The character arrays T and P are often called strings of character.

The string-matching problem can also be seen as the problem of finding all valid shifts with which a given pattern P occurs in a text T.
We say that the pattern P occurs with shift s in the text T if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$.

2.2 Example

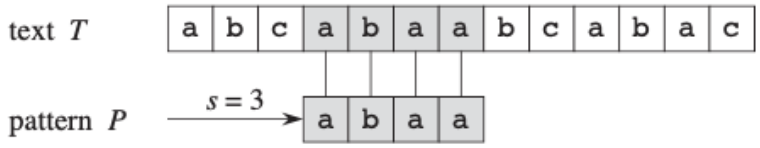

Figure 1: An example of the string-matching problem. Given the text $T = abcabaabcabac$ and the pattern $P = abaa$, we want to find all the occurrences of P in T. The pattern occurs only once in T, starting at shift $s = 3$.

$$\begin{cases} 0 \leq s \leq n - m \\ T[s + 1..s + m] = P[1..m] \\ T[s + j] = P[j] \text{ for } (1 \leq j \leq m) \end{cases}$$

2.3 Practical application

String matching algorithms have many practical applications in different fields:

- Text search: Find the occurrences of words or phrases within documents or long texts.
- Natural language processing: Analysing text to recognise language patterns, lexical analysis and parsing of texts.
- Computational biology: Analysing DNA, RNA and protein sequences to identify biological patterns and correlations.
- Log and structured text analysis: Recognise patterns in large datasets, such as system logs, sensor records, etc.
- Data filtering and analysis: They allow information to be extracted from large volumes of structured and unstructured data.
- Search engines: Working behind the scenes to identify the most relevant matches between user search queries and content.
- Data compression: Used in compression algorithms to find and replace repeated patterns.
- Computer security: Identify signatures and patterns of attacks by detecting byte sequences or malicious behaviour in data.

3 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ values of s.

3.1 Description

The naive string-matching algorithm proceeds sliding a "template" containing the pattern over the text, noting for which shifts all of the characters in the template match the corresponding characters in the text.
The for loop consider each possible shift explicitly, and the if-condition implicitly loops to check corresponding characters position until all position match successfully or until a mismatch is found.

3.2 Pseudocode

Data: Text T of size n and pattern P of size m
Result: All valid shifts with which P occurs in T

```
1 NAIVE-STRING-MATCHER(T,P)
2 n ← T.length
3 m ← P.length
4 for s ← 0 to n − m do
5 |   if P[1..m] = T[s + 1..s + m] then
6 |   |   print "Pattern occurs with shift" s;
7 |   end
8 end
```

3.3 Time complexity

To analyse the time complexity of the algorithm, we can study separately the case in which the pattern occurs only once within the text and the case in which it occurs several times. In the first case we can identify three situations: the case in which the pattern is at the beginning of the text (best-case), the case in which it is at the end of the text (worst-case) and the case in which it is anywhere else in the text (average-case).

- **Best-case:** the pattern is found immediately at the beginning of the text. In this case the complexity is $\Theta(m)$, where m is the length of the pattern.
- **Worst-case:** the pattern is found at the end of the text. In this case the complexity is $\Theta((n-m+1)m)$, where n is the length of the text and m is the length of the pattern.
- **Average-case:** the pattern is found anywhere else in the text. In this case the complexity is $O((n-m+1)m)$, where n is the length of the text and m is the length of the pattern.

In the second case, on the other hand, we find only a single case, since the algorithm is forced to view the entire text anyway (this is similar to the worst-case treated in the previous situation). In fact, in this case the complexity is $\Theta((n-m+1)m)$.

4 String matching with finite automata

Some algorithms, in order to reduce the time complexity of the matching process, use a finite automaton to implement a **preprocessing function** that examines the pattern P. There's in fact a pattern matching technique, the **string matching with finite automata**, that employs finite automata to efficiently search for occurrences of a given pattern within a larger text. These string-matching automata examine each text character exactly once, taking constant time per text character. The matching time used, after preprocessing the pattern to build the automaton, is therefore $\Theta(n)$, where n is the length of the text. Although it is independent of the size of the text, the time to build the automaton can be large if Σ is large. In fact, the time to process the pattern using the function δ is $O(m|\Sigma|)$, where m is the length of the pattern and $|\Sigma|$ is the size of the alphabet.

4.1 Finite automaton

Concept:

- A finite automaton is a simple machine for processing information, used to recognize patterns within input sequences.
- It's a computational model consisting of a finite set of states and transitions between those states based on input symbols.
- The transition function δ defines the ways in which one goes from one state to another.
- In this context, it's used to recognize whether the pattern occurs in the text.

Process:

- Preprocessing: construct a finite automaton that represents the pattern to be matched.
- Matching: use the finite automaton to scan through the text and identify potential matches.

Advantages:

- Efficient for multiple searches of the same pattern within different texts.
- Requires preprocessing only for the pattern.

Implementation:

- Construction of Finite Automaton: this involves building a transition function δ that defines the behavior of the automaton when reading characters of the pattern.
- Matching: traverse the text while updating the automaton's state based on the characters encountered. If the automaton reaches an accepting state, a match is found.

5 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm is a string-matching algorithm that uses finite automata to perform string matching. In fact, the Knuth-Morris-Pratt algorithm improves the running time of the naive algorithm by using information about the pattern itself to avoid some comparisons, using a preprocessing function π.
The key idea is that when a mismatch occurs, the pattern itself contains enough information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

5.1 Description

It preprocesses the pattern to create a "failure function" that determines potential shifts in case of mismatches during pattern matching. This information allows the algorithm to skip unnecessary comparisons, enabling faster pattern identification within the text. The KMP algorithm scans the text, leveraging the failure function to avoid redundant checks, and reports all positions where the pattern is found within the text.

5.2 Pseudocode

Data: Text T of size n and pattern P of size m
Result: All valid shifts with which P occurs in T

```
KMP-MATCHER(T,P)
n ← T.length
m ← P.length
π ← COMPUTE-PREFIX-FUNCTION(P)
q ←0
for i ← 1 to n do
    while q > 0 and P[q + 1] ≠ T[i] do
        q ← π[q];
    end
    if P[q + 1] = T[i] then
        q ← q + 1;
    end
    if q = m then
        print "Pattern occurs with shift" i − m;
        q ← π[q];
    end
end
```

```
Data: Pattern P of size m
Result: Prefix function π
1  COMPUTE-PREFIX-FUNCTION(P)
2  m ← P.length
3  let π[1..m] be a new array;
4  π[1] ← 0;
5  k ← 0;
6  for q ← 2 to m do
7  |  while k > 0 and P[k + 1] ≠ P[q] do
8  |  |  k ← π[k];
9  |  end
10 |  if P[k + 1] = P[q] then
11 |  |  k ← k + 1;
12 |  end
13 |  π[q] ← k;
14 end
15 return π;
```

5.3 Time complexity

Let us analyse the time complexity of the KMP algorithm by examining the preprocessing and matching phases separately.

We know that KMP uses a preprocessing function π, based on the concept of finite automata, but implements it in such a way as to be able to compute the transition function δ efficiently ("on the fly" as needed). In fact, since $\pi[1..m]$ is the array in which the values of the function of the same name are saved, and since $\pi[1..m]$ has only m entries, this allows the KMP to save a factor of $|\Sigma|$ in the preprocessing time. The time complexity of the preprocessing function π of the KMP is therefore $\Theta(m)$.

The time for the matching phase is therefore $\Theta(n)$, as already mentioned in the Section 4 (String matching with finite automata). This is because the preprocessing function allows the algorithm to scroll through the text by examining the characters only once, and consequently the complexity is reduced only to the text size, which is n.

6 Differences between naive string-matching and KMP algorithms

6.1 Preprocessing

Algorithm	Preprocessing time
Naive	0
KMP	$\Theta(m)$

Table 1: Preprocessing time complexity.

6.2 Matching time complexity

Pattern with only one occurrence in the text:

	Matching time		
Algorithm	**Best-case**	**Worst-case**	**Average-case**
Naive	$\Theta(m)$	$\Theta((n-m+1)m)$	$O((n-m+1)m)$
KMP	$\Theta(m)$	$\Theta(n)$	$O(n)$

Table 2: Matching time complexity (case 1).

Pattern with multiple occurrences in the text:

Algorithm	Matching time
Naive	$\Theta((n-m+1)m)$
KMP	$\Theta(n)$

Table 3: Matching time complexity (case 2).

7 Tests and results

Various types of tests were generated to analyse the two algorithms, more precisely 4 types of tests: random tests (tests with different texts and different patterns), tests with the same text of different sizes but same pattern in every test case, tests with the same text but different patterns, tests with an entire chapter of a book as text and a word as pattern.

7.1 Expected performances

Based on the study of the complexities of the two algorithms (Section 6), we can expect them to perform similarly in test cases with small pattern sizes; this is because the time for the preprocessing phase of the KMP (which is $\Theta(m)$) tends to that of the naive (which is 0), while the matching time of the naive (which is $\Theta((n-m+1)m)$) tends to that of the KMP (which is $\Theta(n)$). We can also guess that, for the same pattern size, as the text size increases, the time of the matching phase of the naive increases more than that of the KMP, obviously due to their time complexity. Finally, we can expect the KMP to perform better than the naive in test cases with large pattern sizes.

7.2 Results

Now follows the tests performed. Input data and runtimes are shown, the representation of which is provided both in graphical and tabular form.

7.2.1 Random tests

Test case	Text	n	Pattern	m
1	"abcdefg"	7	"cd"	2
2	"xyxyxy"	6	"xy"	2
3	"bababababa"	10	"aba"	3
4	"racecar"	7	"race"	4
5	"moonlight"	9	"light"	5
6	"patternpattern"	14	"pattern"	7
7	"0123456789012345"	16	"2345"	4
8	"treefrogstreefrogs"	18	"treefrogs"	9
9	"abcdeedcba"	10	"deed"	4
10	"thisisaverylongtextwith nopattern"	32	"pattern"	7
11	"abcdefabcdef"	12	"abcdef"	6
12	"abababababababababababab"	22	"abab"	4
13	"abcdefghi"	9	"ijk"	3
14	"pythonisfun"	11	"is"	2
15	""	0	"pattern"	7
16	"z" * 100	100	"z" * 10	10
17	"abcdefghijklmnopqrstuwxyz"	25	"aeiou"	5
18	"124578915"	9	"69"	2
19	"one"	3	"two"	3
20	"loremipsumdolorsitamet"	22	"ipsumdolor"	10

Table 4: input values in test 1_1

Test case	Text	n	Pattern	m
1	"patternabacabac"	15	"pattern"	7
2	"abacabacpattern"	15	"pattern"	7
3	"abacababacabacababacab"	22	"abacab"	6
4	"abababababababababab abababababab"	32	"abababababab"	12
5	"a" * 1000000 + "b" * 5000	1005000	"b" * 5000	5000
6	"a" * 1000000 + "ab" * 25000 + "a" * 1000000	2050000	"ab" * 25000	50000
7	"ab" * 1000000	2000000	"abab"	4
8	"abcdefghijklmnopqrstu vwxyz"	26	"12345"	5
9	"abcabcabcabcabc"	15	"abc"	3
10	"abcdefghijklmnopqrstu wxyz" * 20000	500000	"pqrs" * 5000 + "uvwxyz" * 5000	50000
11	"0101010101" * 100000	1000000	"01" * 50000	100000
12	"abcdefgh" * 50000	400000	"abcd" * 25000	100000
13	"a1b2c3d4e5f6g7h8i9j0" * 5000	100000	"12345" * 1000 + "j0" * 1000	7000
14	"!@$%&*()_+=" * 25000	325000	"!@$" * 5000	20000
15	"abcdefgh" * 100000	800000	"abcdefgh" * 50000	400000
16	"0123456789" * 200000	2000000	"56789" * 40000 + "23456" * 40000	400000
17	"a" * 1000000 + "b" * 5000	1005000	"b" * 5000	5000
18	"a" * 1000000 + "ab" * 25000 + "a" * 1000000	2050000	"ab" * 25000	50000
19	"ab" * 1000000	2000000	"abab"	4
20	"a" * 1000000 + "ab" * 25000 + "a" * 1000000	2050000	"ab" * 25000	50000
21	"ab" * 1000000	2000000	"abab"	4
22	"0123456789" * 200000	2000000	"01234" * 40000	200000
23	"abcdefghijklmnopqrstu vwxyz" * 50000	1300000	"uvwxyz" * 10000	60000
24	"abcdefghijklmnopqrstu wxyz" * 100000	2500000	"pqrs" * 25000 + "uvwxyz" * 25000	250000
25	"0101010101" * 50000	500000	"01" * 25000	50000

Table 5: input values in test 1_2

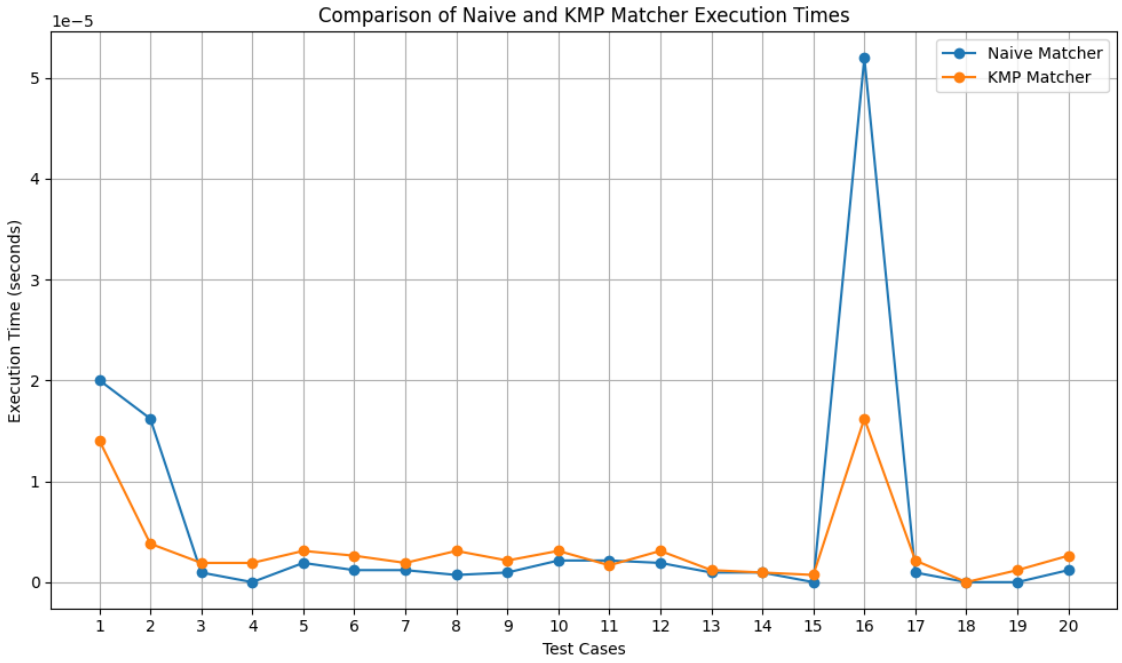

Figure 2: chart test 1_1

Test Case	Naive	KMP			
1	1.7166e-05	1.6212e-05	11	1.1921e-06	1.9073e-06
2	8.8215e-06	6.1989e-06	12	2.1458e-06	2.6226e-06
3	9.5367e-07	1.9073e-06	13	1.1921e-06	9.5367e-07
4	7.1526e-07	2.1458e-06	14	1.9073e-06	2.8610e-06
5	0.0000	1.9073e-06	15	0.0000	9.5367e-07
6	9.5367e-07	1.9073e-06	16	0.0001	2.0027e-05
7	9.5367e-07	2.1458e-06	17	9.5367e-07	1.9073e-06
8	7.1526e-07	2.1458e-06	18	1.1921e-06	7.1526e-07
9	9.5367e-07	9.5367e-07	19	0.0000	0.0000
10	1.9073e-06	1.9073e-06	20	9.5367e-07	3.0994e-06

Figure 3: table test 1_1

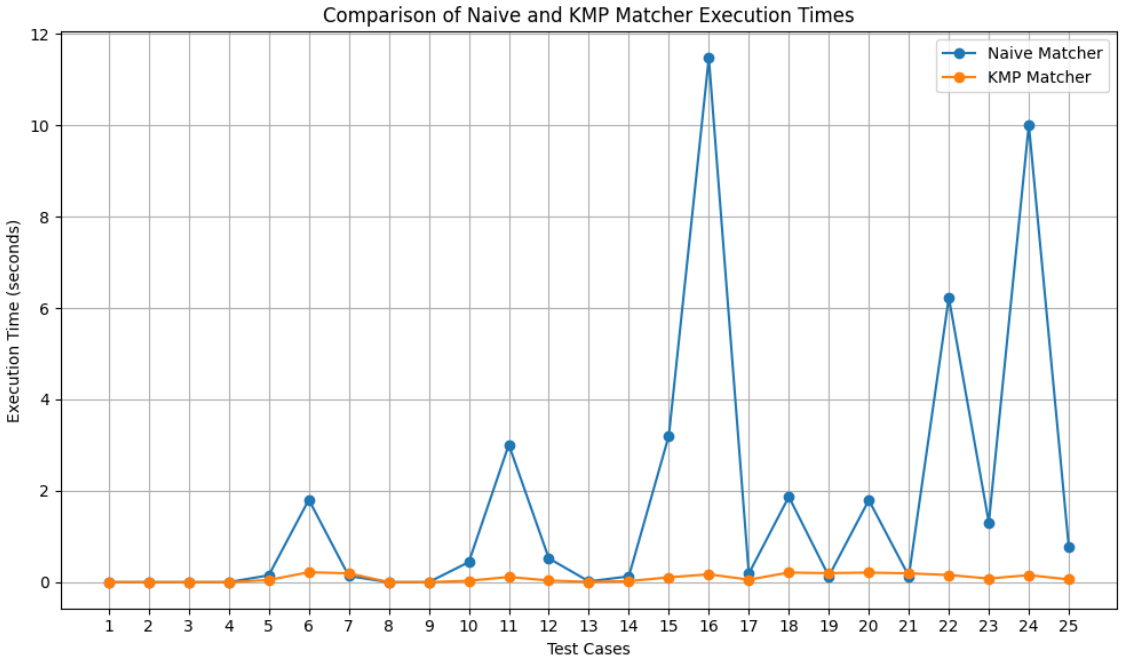

Figure 4: chart test 1_2

Test Case	Naive	KMP			
1	7.1526e-07	2.1458e-06	14	0.1231	0.0217
2	9.5367e-07	2.1458e-06	15	3.0700	0.0995
3	2.1458e-06	1.9073e-06	16	10.3302	0.1707
4	1.9073e-06	4.0531e-06	17	0.1555	0.0484
5	0.1511	0.0470	18	1.6868	0.2073
6	1.8295	0.2056	19	0.1343	0.1950
7	0.1497	0.1906	20	1.7646	0.2070
8	3.8147e-06	3.0994e-06	21	0.1267	0.1918
9	5.0068e-06	5.9605e-06	22	5.8775	0.1552
10	0.4022	0.0303	23	1.2897	0.0786
11	2.9590	0.1066	24	9.0869	0.1537
12	0.6423	0.0362	25	0.7590	0.0531
13	0.0172	0.0054			

Figure 5: table test 1_2

7.2.2 Tests with the same text of different sizes but same pattern in every test case

Text	n	Pattern	m
”0123456789” * 2000	20000	”56789” * 40 + ”23456” * 40	400
”0123456789” * 20000	200000	”56789” * 40 + ”23456” * 40	400
”0123456789” * 200000	2000000	”56789” * 40 + ”23456” * 40	400
”0123456789” * 2000000	20000000	”56789” * 40 + ”23456” * 40	400
”0123456789” * 20000000	200000000	”56789” * 40 + ”23456” * 40	400

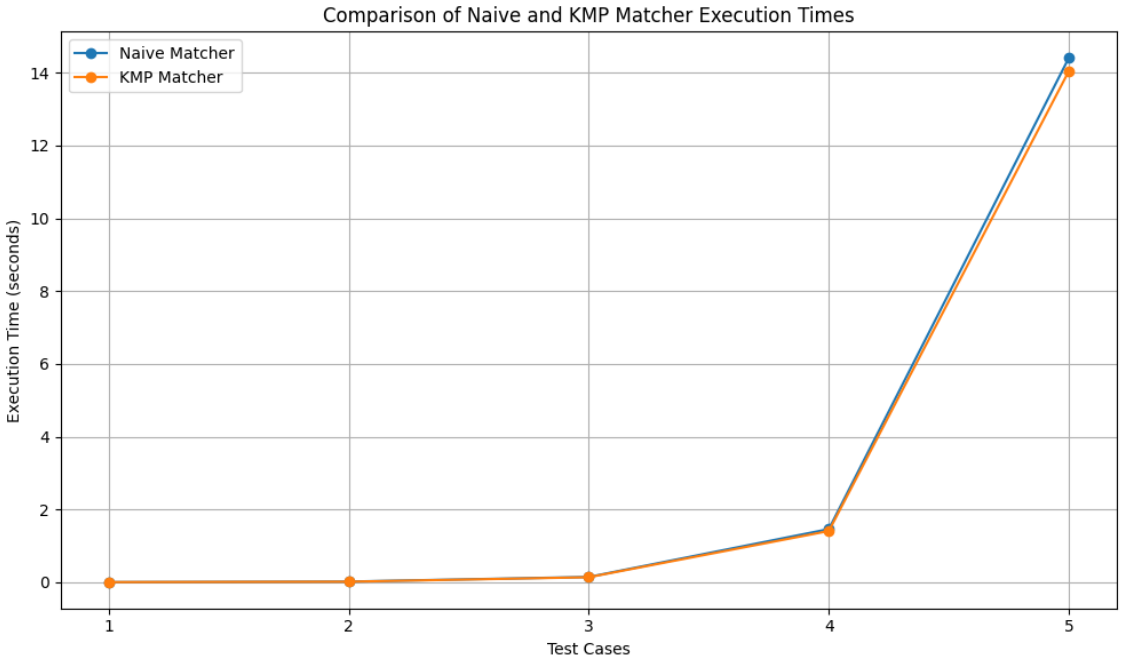

Figure 6: chart test 2_1

Test Case	Naive	KMP
1	0.0015	0.0016
2	0.0140	0.0134
3	0.1428	0.1366
4	1.4054	1.3483
5	14.1260	13.6691

Figure 7: table test 2_1

Text	**n**	**Pattern**	**m**
”0123456789” * 2000	20000	”56789” * 400 + ”23456” * 400	4000
”0123456789” * 20000	200000	”56789” * 400 + ”23456” * 400	4000
”0123456789” * 200000	2000000	”56789” * 400 + ”23456” * 400	4000
”0123456789” * 2000000	20000000	”56789” * 400 + ”23456” * 400	4000
”0123456789” * 20000000	200000000	”56789” * 400 + ”23456” * 400	4000

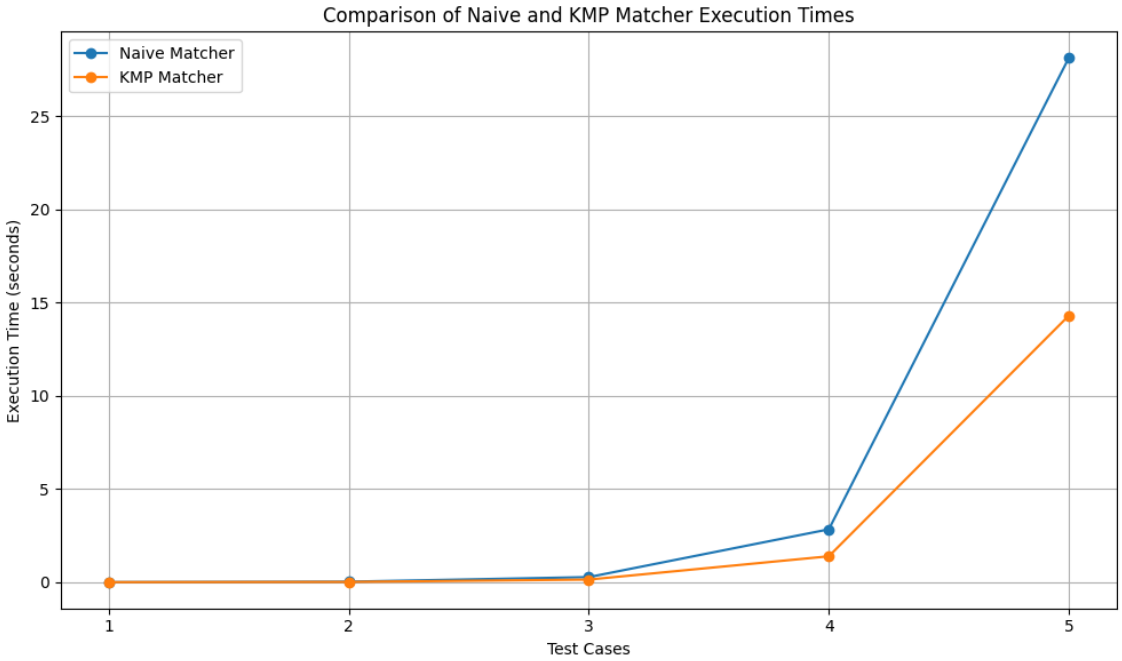

Figure 8: chart test 2_2

Test Case	Naive	KMP
1	0.0025	0.0018
2	0.0291	0.0148
3	0.2981	0.1462
4	2.7937	1.3788
5	27.7417	14.2672

Figure 9: table test 2_2

Text	**n**	**Pattern**	**m**
”0123456789” * 2000	20000	”56789” * 4000 + ”23456” * 4000	40000
”0123456789” * 20000	200000	”56789” * 4000 + ”23456” * 4000	40000
”0123456789” * 200000	2000000	”56789” * 4000 + ”23456” * 4000	40000
”0123456789” * 2000000	20000000	”56789” * 4000 + ”23456” * 4000	40000
”0123456789” * 20000000	200000000	”56789” * 4000 + ”23456” * 4000	40000

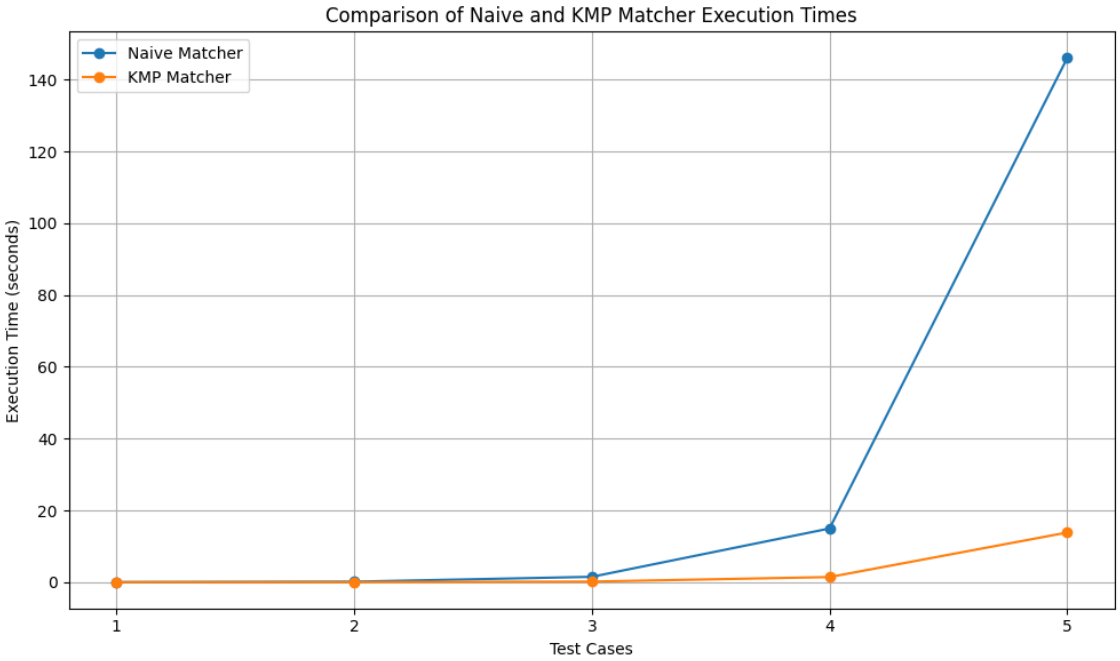

Figure 10: chart test 2_3

Test Case	Naive	KMP
1	8.1062e-06	0.0047
2	0.1586	0.0170
3	1.9318	0.1410
4	15.0266	1.3882
5	148.9970	14.1315

Figure 11: table test 2_3

7.2.3 Tests with the same text but different patterns

Text	n	Pattern	m
”0123456789” * 200000000	2000000000	”56789” * 4 + ”23456” * 4	40
”0123456789” * 200000000	2000000000	”56789” * 40 + ”23456” * 40	400
”0123456789” * 200000000	2000000000	”56789” * 400 + ”23456” * 400	4000
”0123456789” * 200000000	2000000000	”56789” * 4000 + ”23456” * 4000	40000
”0123456789” * 200000000	2000000000	”56789” * 40000 + ”23456” * 40000	400000

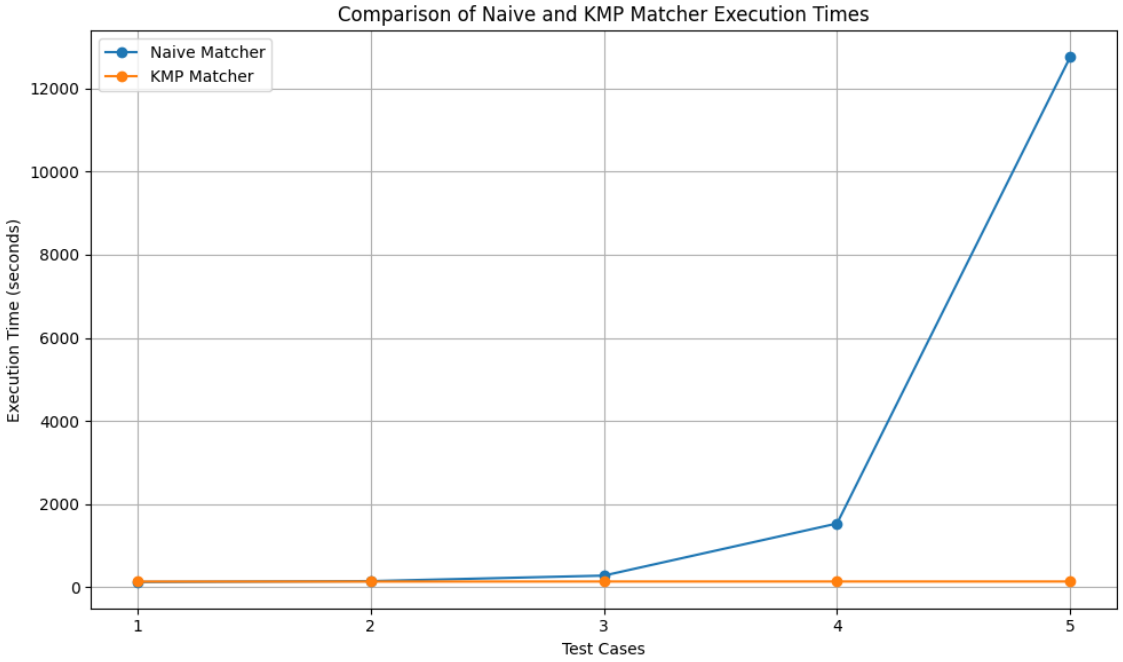

Figure 12: chart test 3_1

Test Case	Naive	KMP
1	129.1751	139.7768
2	150.8726	139.7965
3	283.3899	141.2904
4	1536.3663	141.5856
5	12752.3539	141.5550

Figure 13: table test 3_1

7.2.4 Tests with an entire chapter of a book as text and a word as pattern

First chapter of the book "The Hobbit" by J.R.R. Tolkien as text and the word "hobbit" as pattern.

(http://hbcsni.org/images/9th_Honors_CP_THE_HOBBIT_textbook.pdf)

Text	n
An Unexpected Party	46211

Pattern	m
"hobbit"	6

Test Case	Naive	KMP
1	0.0035	0.0022

Figure 14: table test 4_1

8 Tests discussion and conclusions

Tests confirm what we expected and each test case gives us information about it.
In the random tests (Section 7.2.1) we can see, especially by looking at the graphs (Fig. 2, Fig. 4) that the largest gap between the execution times of the two algorithms occurs in the test cases where the pattern size is largest, e.g.: in test 1_1 the test case 20, while in test 1_2 the test cases 6, 11, 15, 16, 18, 20, 22, 24.
Examining the tests of the second type (Section 7.2.2), we can note that in the cases in which the size of the pattern is moderately "small" (in the tests carried out we speak of the order of hundreds) there is not much difference between the execution times of the naive and the KMP, even as the size of the text increases (the difference remains rather constant, especially in the 2_1 test (Fig. 6), where the times seem to overlap).
In fact, in tests 2_2 and 2_3 (Fig. 8, Fig. 10), the gap between the execution times of the two algorithms grows considerably as the size of the text increases (in these two tests we speak of a pattern size in the order of thousands in the first test, and in the order of tens of thousands in the second).
The final confirmation is provided by test 3_1 (Section 7.2.3) in which we both vary the pattern on the same text with a set size of 200000000, going from an m = 40 to an m = 400000. In fact, in this test we see how the times of the naive "explode" compared to those of the kmp (Fig. 12) .
We then have a "practice" test (Section 7.2.4) in which we test the two algorithms on a chapter of a famous book: the first chapter of "The Hobbit" by J.R.R. Tolkien. Here too we see how, given a pattern of rather small size, the two times resemble each other (Fig. 14).

In conclusion, we can say that the KMP algorithm is more efficient than the naive algorithm in the case of large pattern sizes, and offers a very good solution when performing numerous tests with the same pattern as the algorithm allows us to keep the result of the COMPUTE-PREFIX-FUNCTION(P) (the preprocessing function π that has complexity $\Theta(m)$) and to have a matching phase with complexity always equal to $\Theta(n)$ where n is the size of the text.