Université Libre de Bruxelles

Data Structures and Algorithms

INFO-F-413

String Matching

Author:
Charlotte Nachtegael

Supervisor:
Pr. Jean Cardinal

December 2016

1 Introduction

Looking for patterns in a text is a problem encountered in a lot of situations, such as text-editing programs, in bioinformatics when searching for pattern in the DNA or even when searching for relevant queries with search engines on Internet.

We formalize the string matching problem as follows : we consider a pattern P as an array $P[1...m]$ of length m and a text T as an array $T[1...n]$ of length n, with $m \leq n$. The elements of the arrays are drawn from a finite alphabet Σ with a size $|\Sigma|$. We define that a pattern P occurs with a shift s in the text T if $0 \leq s \leq n-m$ and $P[1...m] = T[s+1...s+m]$. If P occurs in T with a shift s, this shift is qualified **valid**, otherwise it is qualified **invalid**. The string-matching problem is to find all the valid shifts for the pattern P in the text T.

We will present during this report three different strategies for string matching: the Naive one, the Knuth-Morris-Pratt (KMP) algorithm [Knuth, 1977] and the Boyer-Moore (BM) algorithm [Boyer and Moore, 1977]. The notations used are from the book *Introduction to Algorithms, third edition* [Cormen et al., 2009] (CLRS). The aim of this report is to present the worst cases analysis and some average case analysis of the number of comparisons of characters made with these different strategies, proving it by demonstration and back these affirmations with experimental results of original implementations.

2 Necessary terminology and notation

The set of all finite strings is noted Σ^*. The **length** of a string x is denoted $|x|$. A string w is a **prefix** of the string x if $x = wy$ for any $y \in \Sigma^*$ and is noted $w \sqsubset x$. w is a **suffix** of x if $x = yw$ for any $y \in \Sigma^*$ and is noted $w \sqsupset x$. In both cases, $|w| \leq |x|$.

The k^{th} prefix of the pattern $P[1...m]$, $P[1...k]$, is noted as P_k. Similarly the k^{th} prefix of the text T is noted T_k. Thusly, we can note the string matching problem as finding all the shifts s such as $P \sqsupset T_{s+m}$, with $0 \leq s \leq n - m$.

3 Naive algorithm

The naive algorithm is based on the idea of matching all the characters of the pattern for all the possible shifts, of the number of $n-m+1$, until there is mismatch, then the pattern shift from 1 to the right to begin to check the next shift. The computational time is $O(m(n-m))$. The maximum number of comparisons is $m(n-m)$, because you compare all your characters at each possible shift. This could happen when you have all your text with "a" and you pattern composed of at least $(m-1)*$"a". This means that you would have to check all your pattern (m characters) before finding if your patterns occurs with shift s or not.

We implemented a worst case scenario where the text was "a"*43 and the pattern was "ab" and we obtained a number of comparisons equal to $m(n-m)$.

In the CLRS book [Cormen et al., 2009], they proposed an upper bound value for the number of comparisons of characters, for randomly chosen pattern P and text T from an alphabet $\Sigma_d = \{0, 1, ...d-1\}$ with $d \geq 2$, where the comparison stops when a mismatch occurs, as follows:

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1)$$

This affirmation is based on the expected number of trials before success :

$$E[X] = \frac{1-(1-p)^n}{p},$$

where p is the probability of success and E[X] the sum of the probabilities of the success after the ith trial.In this case, the success is defined as the fact that, after $i-1$ matches, the ith character does not, with a probability of $1-\frac{1}{d}$. The final success case is when all the characters match, so all m comparisons were made. Thusly, we obtain :

$$E[X] = \frac{1 - \left(1 - \left(1 - \frac{1}{d}\right)\right)^m}{1 - \left(\frac{1}{d}\right)} \tag{1}$$

$$E[X] = \frac{1 - d^{-m}}{1 - d^{-1}} \tag{2}$$

This expected number of comparisons is used for all possible shift, so $(n - m + 1)$ times, this is why the total expected number is $(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}}$. To complete the proof, we must, prove that $\frac{1 - d^{-m}}{1 - d^{-1}}$ is less or equal to 2.

As said in the beginning, $d \geq 2$, so this means that if $d = 2$, $1 - d^{-1} = 0.5$ and as d increases, so does $1 - d^{-1}$; and $1 - d^{-m}$ will always be less than 1. Consequently $\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2$ (1/0.5 for $d = 2$ and always decreasing as d increases so the denominator increases and the numerator is asymptotically close to 1), confirming the affirmation that $(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$ for the average number of comparisons for a random text and pattern.

We implemented the naive algorithm with text of different lengths from 1,000 to 10,000 and patterns of length 100, created randomly from a alphabet $\Sigma = \{A, G, T, C\}$. We averaged the results over 10 patterns of lengths 100. The results are plotted with the upper bound. We can see that the upper bound is largely overestimated when the text length increases(Fig 1). However, we can distinguish that when the text length is small, the upper bound is quite close to the real number of comparisons.

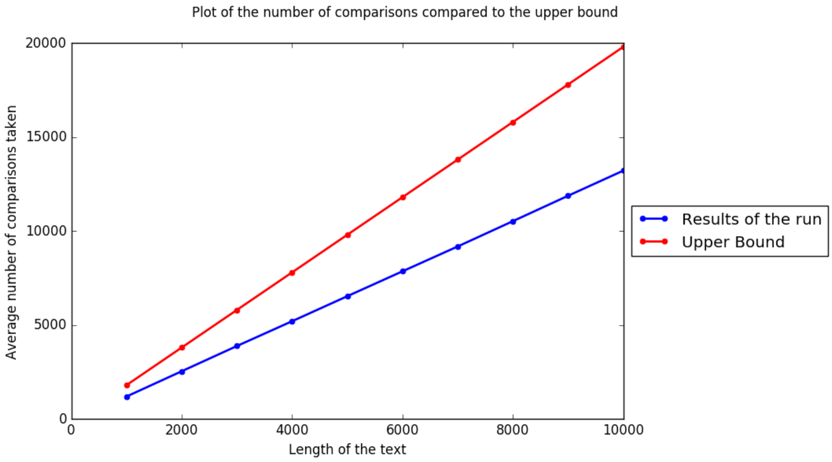

Figure 1: Average number of comparisons for different lengths of text averaged over 10 patterns of length 100 for the naive algorithm

4 KMP algorithm

The Knuth-Morris-Pratt algorithm is based on the construction of a prefix table designated as π, with $\pi[i] = max\{k : k < i \text{ and } P_k \sqsupset P_i\}$. This value $\pi[i]$ is used to shift after mismatch towards a place where a string of characters equal to the prefix of the pattern P were already matched, so as to avoid in-between invalid shifts. The worst case scenario is to obtain $2n$ comparisons. This is due to the fact that at each iteration of the for loop (line 6), we either increase q or slide the pattern right. These two events can occur at most n times, so the loop takes at most $2n$ times [Knuth, 1977]. This worst case happens in the same condition as described earlier for the naive algorithm, with a text composed only of "a" characters and a pattern equal to "ab". We confirmed with an implementation that the number of comparisons was indeed equal to $2n$.

Algorithm 1 KMP string matching algorithm

```
1: procedure KMP MATCHER(T, P)
2:     n ← T.length
3:     m ← P.length
4:     π ← COMPUTE − PREFIX − FUNCTION(P)
5:     q ← 0
6:     for i ← 0 to n do
7:         while q > 0 and P[q + 1] ≠ T[i] do
8:             q ← π[q]
9:         end while
10:        if P[q + 1] == T[i] then
11:            q ← q + 1
12:        end if
13:        if q == m then
14:            print"Pattern occurs with shift" i - m
15:            q ← π[q]
16:        end if
17:    end for
18: end procedure
```

The implementation was done as previously described for the naive algorithm to prove the correctness of the upper bound. We can see that, again, the upper bound is much bigger than the concrete number of comparisons (Fig 2). This is due here to the fact that, despite the randomness of both the text and the pattern, the prefix of the pattern can be found along the rest of the pattern, allowing a more efficient shift along the text. The difference between the upper bound and the results is more marked than with the naive algorithm. This is the proof of the efficiency of the KMP algorithm compared to the naive method. Indeed, we can analyze the complexity of the KMP-matcher with an amortized analysis. The potential function is in function of q. The initial value of q is 0 (line 5). The value of q decreases at line 8 and 15 since $q < \pi[q]$ and since $\pi[q] \geq 0$, the condition that the potential function is always greater or equal than 0 is respected. The value of q is incremented by 1 at line 11 at most once by for loop. The cost of the while loop at line 7-9 is paid by the decrease of the value of q. The cost of the for loop is paid by the incrementation at line 11, so that the for loop operation cost constant time $O(1)$. As the for loop is repeated n times, the cost of only the searching part of the KMP-matcher is $O(n)$.

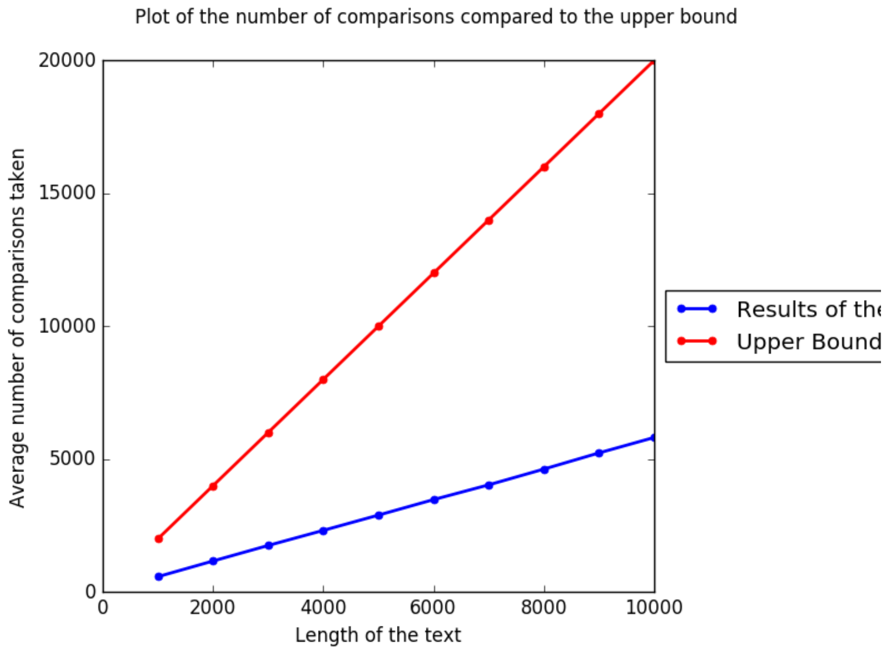

Figure 2: Average number of comparisons for different lengths of text averaged over 10 patterns of length 100 for the KMP algorithm

5 BM algorithm

The Boyer-Moore algorithm is based on the idea of matching from right to left with a shift calculated either with a suffix rule (inspired by the prefix rule in the KMP algorithm) or the bad character rule, where you look if the last bad match character can be found in the pattern at the most left position. The shift chosen is the biggest one. The worst case analysis by Knuth [Knuth, 1977] bounded at $7n$ comparisons. However, Colussi found another upper bound of $4n$ [Colussi, 1991]. While Guibas conjectures for $2n$ comparisons [Guibas, 1980], the current used upper bound number of comparisons is $3n$ for a non-periodic pattern [Cole, 1994].

We implemented as previously described to check the veracity of the upper bound. Again, we constated that the upper bound is largely overestimated (Fig 3). However, the difference here is even more marked than with the KMP algorithm. This means that the efficiency of the BM algorithm with random text and pattern is quite high. This result is due to the fact that BM algorithm shifts according to two different rules, making it able to shift as much as possible.

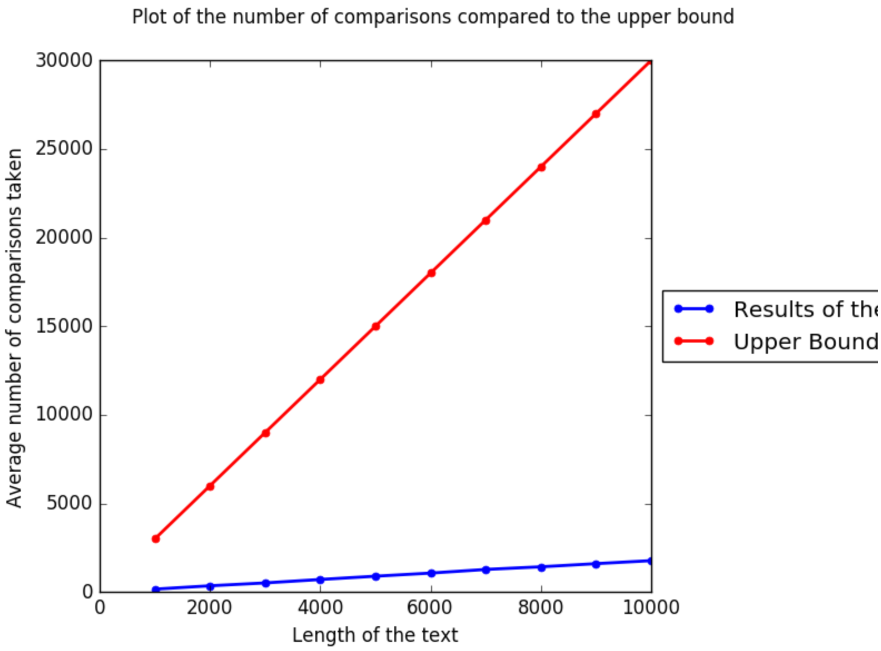

Figure 3: Average number of comparisons for different lengths of text averaged over 10 patterns of length 100 for the BM algorithm

6 Comparison of running time

We compared the time taken by different string matching algorithm. The text was of a length 10,000 and we modified the lengths of the pattern, from 10 to 500. We ran 10 different runs with 10 random words for each length of pattern, all created from an alphabet $\Sigma = \{0, 1\}$.

We compared the naive, the Rabin-Karp, the Knuth-Morris-Pratt and the Boyer-Moore algorithms, respectively with average complexity of $O(mn)$, $O(m + n)$, $O(m + n)$ and $O(m/n)$ in the best case. The Rabin-Karp algorithm is based on a rolling hash function. The pattern and the current part of the text of the same length as the pattern is hashed and compared as two numbers. A comparison of characters is only done when the two numbers are equal. The rolling hash function allows to compute the hash of the next text window in a constant time.

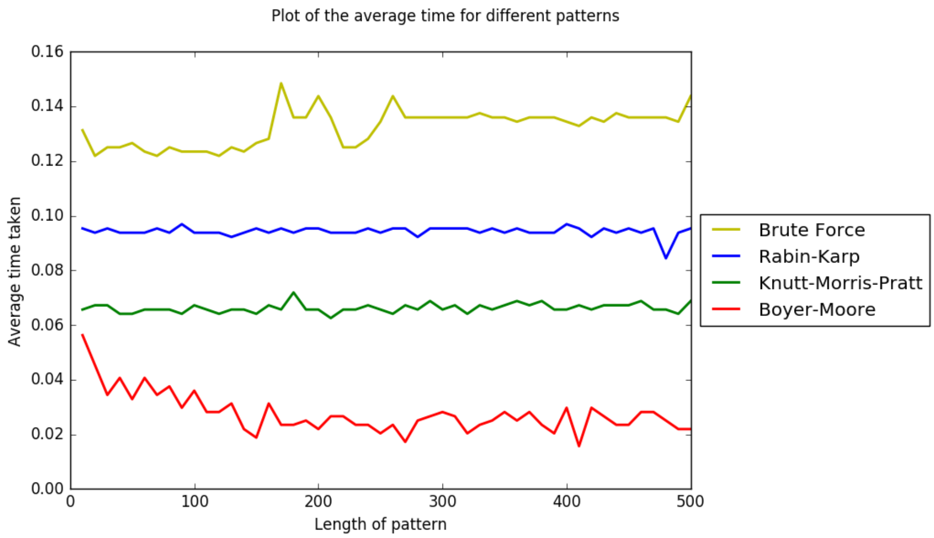

Figure 4: Average time taken by the different string matching algorithms for different lengths of pattern. The results were obtained over 10 runs of 10 random words for each length of pattern confronted to a random 10,000 text, created from a binary alphabet.

We can indeed observe that the naive algorithm performs the worst and that the Boyer-Moore performs the best (Fig 4). Even though the Rabin-Karp and the KMP algorithms have the same time complexity, in practice, we observe that the KMP performed better. This could be due to the fact that, even if the Rabin-Karp compares the pattern and the text only when a match is found with the two strings hashed. However, we could have what is called a **spurious hit**, a false positive hit obtained because of a collision of the hash function. This situation can be avoided by choosing the q parameter of the Rabin-Karp algorithm (a prime number usually chosen to be greater than the length of the pattern) is chosen randomly.

Rabin-Karp and KMP are also fairly linear, whereas we can see that the complexity of the naive algorithm increase with the length of the pattern and the BM complexity decreases with the length of the pattern. These results concords with the fact that the BM algorithm is the most used string matching algorithm in general compared to the other ones.

References

Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. ISSN 0001-0782. doi: 10.1145/359842.359859.

Richard Cole. Tight bounds on the complexity of the boyermoore string matching algorithm. *SIAM Journal on Computing*, 23, 10 1994. doi: 10.1137/s0097539791195543. URL `http://gen.lib.rus.ec/scimag/index.php?s=10.1137/s0097539791195543`.

Livio Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95(2):225 – 251, 1991. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/0890-5401(91)90046-5. URL `http://www.sciencedirect.com/science/article/pii/0890540191900465`.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

Andrew M. Guibas, Leo J.; Odlyzko. A new proof of the linearity of the boyermoore string searching algorithm. *SIAM Journal on Computing*, 9, 11 1980. doi: 10.1137/0209051. URL `http://gen.lib.rus.ec/scimag/index.php?s=10.1137/0209051`.

Jr. James H.; Pratt Vaughan R. Knuth, Donald E.; Morris. Fast pattern matching in strings. *SIAM Journal on Computing*, 6, 06 1977. doi: 10.1137/0206024.

Code

```
"""
INFO-F413 : Data Structures and Algorithms
All Algorithms for the Project 2016
Theme : String Matching
Naive, Rabin-Karp, Knutt-Morris-Pratt and Boyer-Moore
"""

import time
import numpy as np
import matplotlib.pyplot as plt


def naive(text, pattern):
    nb_comp = 0
    for i in range(len(text)-len(pattern)):
        for j in range(len(pattern)):
            nb_comp += 1
            if text[i+j] != pattern[j]:
                break
            elif j == len(pattern) - 1:
                print("Pattern occurs with shift", i - 1)
    return nb_comp


def Rabin_Karp(text, pattern, d, q):
    """
    :param text: string where you look for the pattern
    :param pattern: string you are looking for
    :param d: number, radix, generally the number of characters used
    :param q: prime number
    """
    n = len(text)
    m = len(pattern)
    h = d ** (m - 1) % q
    p = 0
    t = 0

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    for s in range(n - m + 1):
```

```
            # hit
            if p == t:
                # check if it is a spurious hit or not
                if pattern == text[s:]:
                    print("The pattern occurs at shift", s-1)
            if s < n - m:
                t = (d*(t - ord(text[s])* h) + ord(text[s + m])) % q


def Knutt_Morris_Pratt(text, pattern):
    n = len(text)
    m = len(pattern)
    prefix = prefix_function(pattern)

    nb_comp = 0
    q = -1  # number of character matched
    for i in range(n):
        # no match for next character when a match was already found
        while q > -1 and pattern[q + 1] != text[i]:
            nb_comp += 1
            q = prefix[q]
        # match with next character
        if pattern[q + 1] == text[i]:
            nb_comp += 1
            q = q + 1
        # complete match
        if q == m - 1:
            # print("Pattern occurs with shift", i - m + 1)
            q = prefix[q]
    nb_comp += 1
    return nb_comp


def Boyer_Moore(text, pattern, alphabet):
    n = len(text)
    m = len(pattern)
    delta = last_occurrence(pattern, alphabet)
    gamma = good_suffix(pattern)

    s = 0
    nb_comp = 0

    while s <= n - m:
        j = m - 1
```

```
        # good match
        while j > 0 and pattern[j] == text[s + j]:
            j = j - 1
            nb_comp += 1
        #complete march
        if j == 0:
            print("Pattern occurs at shift", s)
            s = s + gamma[0]
        else:
            nb_comp += 1
            # if gamma[j] > j - delta[text[s + j]]:
            # print("Shift good suffix")
            # else:
            # print("Shift bad character")
            s = s + max(gamma[j], j - delta[text[s + j]])
    return nb_comp


def prefix_function(pattern):
    """
    Create a array where the ith entry is the longest suffix to i also prefix of the pattern
    """
    m = len(pattern)
    prefix = [-1] * m
    k = -1
    for i in range(1, m):
        # next character of the pattern does not match the current
        while k > -1 and pattern[k + 1] != pattern[i]:
            k = prefix[k]
        if pattern[k + 1] == pattern[i]:
            k = k + 1
        prefix[i] = k
    return prefix


def good_suffix(pattern):
    """
    Create a suffix table where you can find the next left motif similar to the suffix of the pattern
    """
    m = len(pattern)
    pi = prefix_function(pattern)
    pattern_prime = pattern[::-1]
```

```
    pi_prime = prefix_function(pattern_prime)

    gamma = []

    for j in range(m):
        gamma.append(m - 1 - pi[m - 1])

    for l in range(m):
        j = m - 1 - pi_prime[l] - 1

        if gamma[j] > l - pi_prime[l]:
            gamma[j] = l - pi_prime[l]

    return gamma


def last_occurrence(pattern, alphabet):
    """
    Create a dictionary with the last occurrence (left most) of the
        character in the pattern or -1 if the character does not occur
        in the pattern
    """
    m = len(pattern)
    delta = {}

    for letter in alphabet:
        delta[letter] = -1

    for j in range(m):
        delta[pattern[j]] = j

    return delta


def create_random_dico(alphabet):
    """
    Create a dictionnary of differents length pattern with 10 words
        each
    """
    dico = {}
    for i in range(10, 510, 10):
        dico[i] = []
        while len(dico[i]) < 10:
            word = np.random.choice(alphabet, i)
```

```
            word = ''.join(word)
            if word not in dico[i]:
                dico[i].append(word)
    return dico


def create_random_text(alphabet):
    """
    Create a list of random texts of different lengths
    """
    random_list = []
    for i in range(1000, 11000,1000):
        text = np.random.choice(alphabet,i)
        text = ''.join(text)
        random_list.append(text)

    return random_list


def main(text, dico, alphabet):
    """
    Run the algorithm to calculate the average time taken for the
        matching
    """

    #create the prime list
    list_primes= []
    with open("primes.txt", "r") as primes:
        for line in primes:
            line = line.strip().split(' ')
            for ele in line:
                if ele.isdigit():
                    list_primes.append(int(ele))

    current_algo = []
    for size in sorted(dico):
        current_size = []

        for word in dico[size]:
            t = time.process_time()
            naive(text, word)
            elapsed = time.process_time() - t
            current_size.append(elapsed)
```

```
        current_algo.append(np.longfloat(np.mean(np.array(current_size))))

    time_algo = np.array([current_algo])

    current_algo = []
    for size in sorted(dico):
        current_size = []
        for word in dico[size]:
            q = np.random.choice(list_primes)
            while q < len(word):
                q = np.random.choice(list_primes)
            t = time.process_time()
            Rabin_Karp(text, word, len(alphabet), q)
            elapsed = time.process_time() - t
            current_size.append(elapsed)
        current_algo.append(np.longfloat(np.mean(np.array(current_size))))

    time_algo = np.append(time_algo, [current_algo], axis=0)

    current_algo = []
    for size in sorted(dico):
        current_size = []

        for word in dico[size]:
            t = time.process_time()
            Knutt_Morris_Pratt(text, word)
            elapsed = time.process_time() - t
            current_size.append(elapsed)
        current_algo.append(np.longfloat(np.mean(np.array(current_size))))

    time_algo = np.append(time_algo, [current_algo], axis=0)

    current_algo = []
    for size in sorted(dico):
        current_size = []
        for word in dico[size]:
            t = time.process_time()
            Boyer_Moore(text, word, alphabet)
            elapsed = time.process_time() - t
            current_size.append(elapsed)
```

```
            current_algo.append(np.longfloat(np.mean(np.array(
                current_size))))

        time_algo = np.append(time_algo, [current_algo], axis=0)

        return time_algo


def plot_all(average_runs):
    """
    Plot the different average times obtained for the algorithms
        fordifferent length patterns
    """
    names = ('Brute Force', 'Rabin-Karp', 'Knutt-Morris-Pratt', '
        Boyer-Moore')

    fig = plt.figure()
    ax = plt.axes()
    colors = ['r', 'g', 'b', 'y']
    list_names = list(names)
    x = np.arange(10, 510, 10)

    for algo in average_runs:
        ax.plot(x, algo, color=colors.pop(), linewidth=2, label=
            list_names.pop(0))
    fig.suptitle('Plot of the average time for different patterns')
    ax.set_xlim([0, 500])
    ax.set_xlabel('Length of pattern')
    ax.set_ylabel('Average time taken')

    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])

    # Put a legend to the right of the current axis
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

    plt.show()


def plot_comparison(average_list,upperbound_list):
    """
    Plot the average of comparisons confronted to the upper bound
        limit
    """
```

```
    fig = plt.figure()
    ax = plt.axes()
    colors = ['r', 'b']
    names = ['Results of the run','Upper Bound']
    x = np.arange(1000,11000,1000)

    ax.plot(x,average_list,marker='.', markersize = 10,color=colors.
        pop(), linewidth=2, label=names.pop(0))
    ax.plot(x, upperbound_list, marker='.',markersize = 10,color=
        colors.pop(), linewidth=2, label=names.pop(0))
    fig.suptitle('Plot of the number of comparisons compared to the
        upper bound')
    ax.set_xlim([0, 10000])
    ax.set_xlabel('Length of the text')
    ax.set_ylabel('Average number of comparisons taken')

    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])

    # Put a legend to the right of the current axis
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

    plt.show()


def average_time():
    """
    Count the average time taken by the different algorithms, average
        on 10 runs and plot them all together
    """
    alphabet = ['0', '1']

    # 100 000 characters
    text = ''
    for i in range(100000):
        text += np.random.choice(alphabet)

    average_runs = []

    for i in range(10):
        dico = create_random_dico(alphabet)
        average_runs.append(main(text, dico, alphabet))
    average_runs = np.mean(np.array(average_runs),axis=0)

```

```
    plot_all(average_runs)


def number_comparisons():
    """
    Count the average number of comparisons for different algorithms
        and plot it against their upper bound limit
    """
    alphabet=['A','G','C','T']

    text_list = create_random_text(alphabet)

    dico = []
    while len(dico) < 10:
        word = np.random.choice(alphabet, 100)
        word = ''.join(word)
        if word not in dico:
            dico.append(word)

    average_list = []
    upper_bound_list = []
    for text in text_list:
        sum = 0
        for word in dico:
            sum += naive(text,word)
        average_list.append(sum/len(dico))
        upper_bound_list.append(2*(len(text)-100 + 1))
    plot_comparison(np.array(average_list),np.array(upper_bound_list)
        )

    average_list = []
    upper_bound_list = []
    for text in text_list:
        sum = 0
        for word in dico:
            sum += Knutt_Morris_Pratt(text,word)
        average_list.append(sum/len(dico))
        upper_bound_list.append(2*len(text))
    plot_comparison(np.array(average_list),np.array(upper_bound_list)
        )

    average_list = []
    upper_bound_list = []
    for text in text_list:
```

```
        sum = 0
        for word in dico:
            sum += Boyer_Moore(text,word, alphabet)
        average_list.append(sum/len(dico))
        upper_bound_list.append(3*len(text))
    plot_comparison(np.array(average_list),np.array(upper_bound_list)
        )


    # worst case scenario
    worst_text = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    worst_pattern="ab"
    print("Worst case - simulation :",naive(worst_text,worst_pattern)
        ,"Bound :",len(worst_pattern)*(len(worst_text)-len(
        worst_pattern)))
    print("Worst case - simulation :", Knutt_Morris_Pratt(worst_text,
         worst_pattern), "Bound :",2*len(worst_text))
    print("Worst case - simulation :", Boyer_Moore(worst_text,
        worst_pattern,alphabet), "Bound :",3*len(worst_text))


if __name__ == "__main__":
    number_comparisons()
    average_time()
```