String Matching Experiment Report

李胜锐 2017012066

1 Experiment Environment

Operating System: macOS 10.15.2

Processor: 2.6 GHz 6-Core Intel Core i7

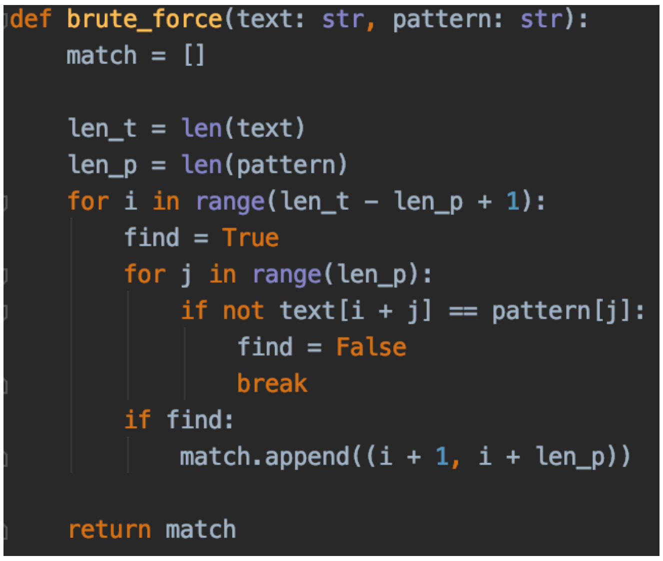
Memory: 16GB

Language: python 3.7

IDE: PyCharm

2 Algorithm Analysis

2.1 Brute Force

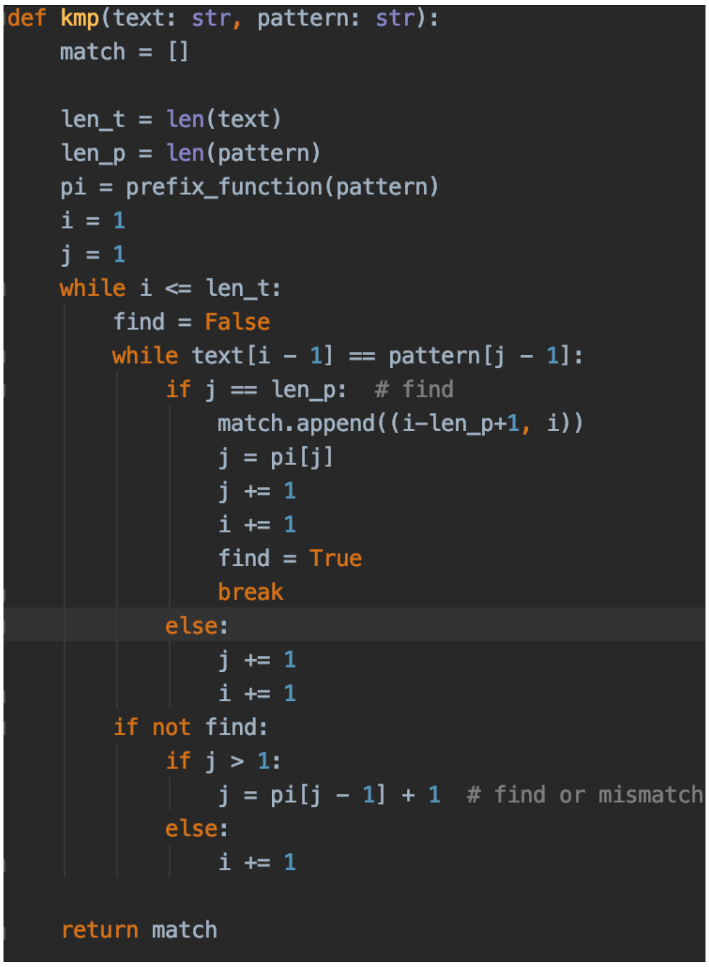


```python
def brute_force(text: str, pattern: str):
    match = []

    len_t = len(text)
    len_p = len(pattern)
    for i in range(len_t - len_p + 1):
        find = True
        for j in range(len_p):
            if not text[i + j] == pattern[j]:
                find = False
                break
        if find:
            match.append((i + 1, i + len_p))

    return match
```

This is a naïve method. It shifts pattern under target text one by one. The worst time complexity can reach to Θ(n(m-n+1)). However, if a char fails to match, the pattern will immediately shift, without comparing the rest chars. Thus, since the whole Alphabet is very large, if the pattern and the corresponding sub-text can't match, comparison will end almost at the beginning. In fact, the T(n) won't be very large.

$$T(n) = \Theta(n+m)$$

2.2 KMP



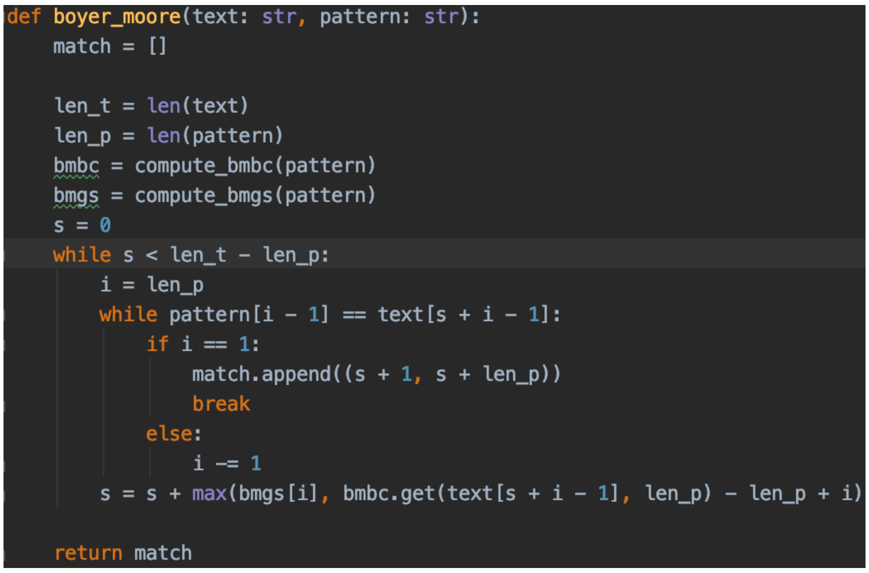
```
def kmp(text: str, pattern: str):
    match = []

    len_t = len(text)
    len_p = len(pattern)
    pi = prefix_function(pattern)
    i = 1
    j = 1
    while i <= len_t:
        find = False
        while text[i - 1] == pattern[j - 1]:
            if j == len_p:  # find
                match.append((i-len_p+1, i))
                j = pi[j]
                j += 1
                i += 1
                find = True
                break
            else:
                j += 1
                i += 1
        if not find:
            if j > 1:
                j = pi[j - 1] + 1  # find or mismatch
            else:
                i += 1

    return match
```

KMP algorithm will first calculate the prefix function, which will cost time Θ(m).

Then, when matching fails, it will move directly to the position of the next valid prefix, saving a lot of time. The time used in match will be Θ(n).

$$T(n) = \Theta(n+m)$$

2.3 Strassen Multiply Algorithm



```python
def boyer_moore(text: str, pattern: str):
    match = []

    len_t = len(text)
    len_p = len(pattern)
    bmbc = compute_bmbc(pattern)
    bmgs = compute_bmgs(pattern)
    s = 0
    while s < len_t - len_p:
        i = len_p
        while pattern[i - 1] == text[s + i - 1]:
            if i == 1:
                match.append((s + 1, s + len_p))
                break
            else:
                i -= 1
        s = s + max(bmgs[i], bmbc.get(text[s + i - 1], len_p) - len_p + i)

    return match
```

This method consists of three parts: Calculating good-suffix function (bmGs), calculating bad-character function (bmBc) and matching. The key point of this method lies in choosing the maximal shifting number between bmGs and bmBc. In this way, the pattern is able to shifts as more as possible.

The preprocessing time cost is Θ (m+|Σ |).

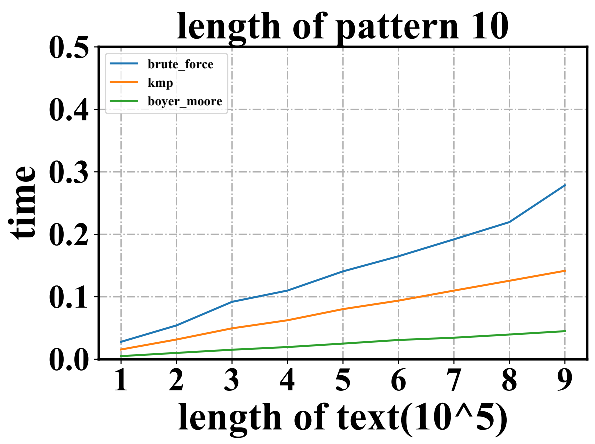
The matching time differs a lot. For the best case, it is Ω(n/m). For the worst case, it is Ω(mn). And the average time is Ω(n).

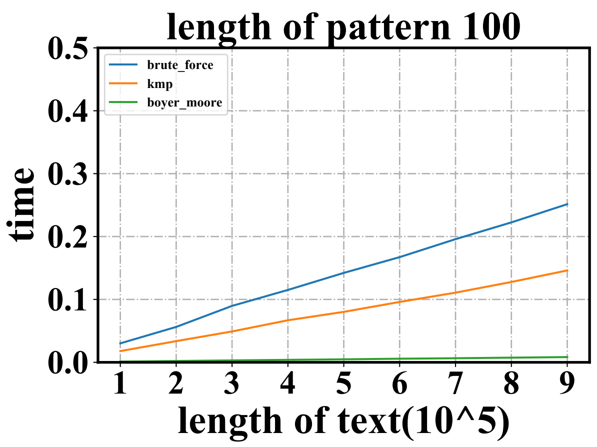
$$T(n) = \Theta(m+|\Sigma|+n)$$

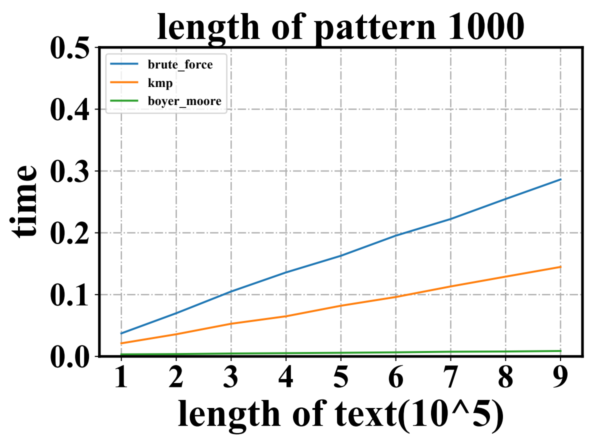
3 Result Analysis

3.1 Comparison of three methods

I generate a random text and insert the random pattern into ten different places into the text. I compare the time cost under different length of text and pattern. The results are as following.







I set the length of text between 10,000 and 100,000. And I also set the length of pattern as 10, 100, 1000. The result can be summarized as:

1. All three methods have linear time complexity with **length of text**.
2. **Length of pattern** has little impact on Brute Force and KMP. However, it influences Boyer Moore method a lot. With longer **length of pattern,** time complexity of Boyer Moore method will decrease, which is a very excellent property.
3. Boyer Moore method is the fastest method, which is much better than the others. KMP is nearly twice as faster as Brute Force.

The following table is the result in larger range of **length of text**.

Pattern		10		
method		Brute Force	KMP	Boyer Moore
text	100	5.9128E-05	4.2915E-05	3.8862E-05
	1,000	3.7789E-04	1.8191E-04	8.4877E-05
	10,000	2.8548E-03	1.5969E-03	5.4312E-04
	100,000	2.8164E-02	1.5393E-02	5.1842E-03
	1,000,000	2.8490E-01	1.5591E-01	5.1694E-02
	10,000,000	3.0058E+00	1.5514E+00	5.3672E-01
Pattern		100		
method		Brute Force	KMP	Boyer Moore
text	100	5.2691E-04	4.2105E-04	3.2902E-04
	1,000	9.6321E-04	6.2680E-04	3.9721E-04
	10,000	4.8301E-03	2.6159E-03	4.7183E-04
	100,000	5.2975E-02	2.8575E-02	2.1918E-03
	1,000,000	2.9779E-01	1.5304E-01	9.9151E-03
	10,000,000	2.9252E+00	1.5161E+00	9.2929E-02
Pattern		1000		
method		Brute Force	KMP	Boyer Moore
text	100	5.3420E-03	4.4219E-03	3.4592E-03
	1,000	5.6438E-03	4.3490E-03	3.1381E-03
	10,000	8.6532E-03	5.3649E-03	2.5780E-03
	100,000	3.6774E-02	1.9233E-02	3.1641E-03
	1,000,000	3.4064E-01	1.6001E-01	1.0555E-02
	10,000,000	3.3863E+00	1.5721E+00	8.2746E-02

4 Conclusion

In this experiment, I implement three string matching algorithms. By comparing implementation time, I discover Boyer Moore performs best and Brute Force performs worst.