[redacted]

Setup

- Generating **Random Text** String:
  - I used Java's Random class to generate a random string of 10,000 characters – that included alphabets, numbers, and special characters.
  - Code: The function takes in the length of the random string desired, generates and returns it.

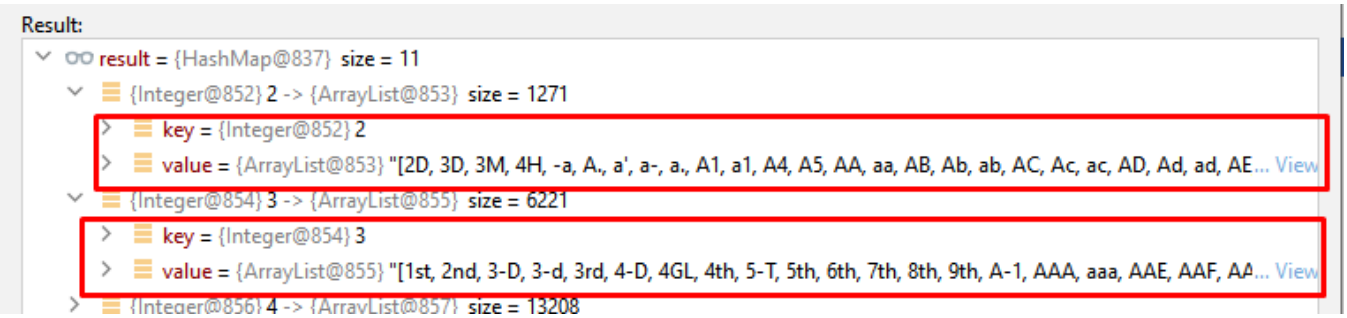
```
private static String getRandomString(int length) {
    int leftLimit = 48, rightLimit = 127; // ASCII printable
    Random random = new Random();
    return random.ints(leftLimit, rightLimit)
            .limit(length)
            .collect(StringBuilder::new,
                     StringBuilder::appendCodePoint,
                     StringBuilder::append)
            .toString();
}
```

- Generating **Random Pattern** Strings:
  - For each m, I generated 1000 random patterns using the same function above. Then calculated the average running time for finding each pattern in the text.
    - Eg: m = 2 -> ig, 5j, !p. AB, Kp, …. (1000 random patterns)
    - Eg: m = 3 -> oie, 19f, ;rt, qec, …. (1000 random patterns)
  - Since these pattern strings were generated at random, some were present in the Text string, and some weren't.
  - Since the code tested the execution times for 1000 random patterns, the average elapsed time was a good indicator of the real-world execution time of each algorithm.

- Reading speeches.txt file for **English Words** Matching
  - I downloaded and stored the speeches.txt file in the same directory as the code.
  - The code used Java's `Files.readAllBytes()` function

```
byte[] bytes = Files.readAllBytes(Paths.get("project2/speeches.txt"));
String text = new String(bytes);
```

- Reading words from words1.txt file for **English Words** Matching
  - I used the same words1.txt file from Assignment 1.
  - The code read the words using `BufferedReader` and stored the words in a HashMap of length to List of words. Screenshot from IntelliJ:



- From each M above, the code randomly samples 1000 words from the list and calculates the average over multiple runs.
- Code:

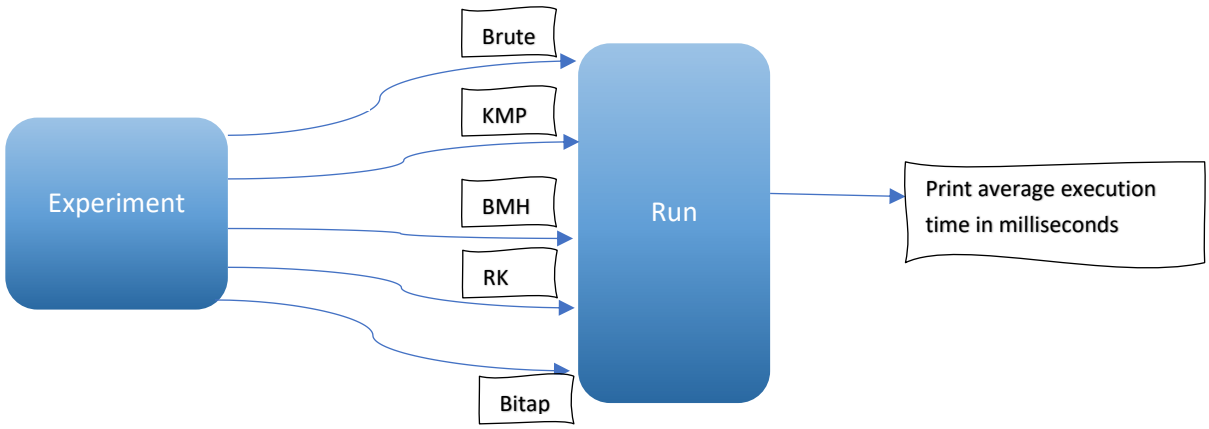
```
Random random = new Random();
for (int l : lengths) {
    int numberOfWords = wordMap.get(l).size();
    for (int i = 0; i < 1000; i++) {
        m[i] = wordMap.get(l).get(random.nextInt(numberOfWords)); // choose a
random word from list
    }
    experiment(m[0].length(), m, text); // experiment 1000 times for each m
}
```

- Calculating the Execution Time
  - I used Java's `System.nanoTime()`
  - Code: The function takes in the algorithm to run. Executes the algorithm for the given 1000 pattern strings. And prints the average execution time.

```
private static void run(String[] M, String T, String algo, Matcher matcher) {
    double totalElapsed = 0;
    for (String P : M) {
        long start = System.nanoTime();
        int res = matcher.match(T, P);
        long finish = System.nanoTime();
        totalElapsed += finish - start;
    }
    totalElapsed /= 1000.0;
    System.out.printf(algo +": %.5f microseconds", totalElapsed / 1000.0);
}
```

  - The Above run method is called for all the five algorithms by passing the desired algorithm object. All the algorithm class extend the Matcher class and use inheritance

```
private static void experiment(int mLength, String[] M, String T) {
    run(M, T, "Brute", brute);
    run(M, T, "KMP  ", kmp);
    run(M, T, "BMH  ", bmh);
    run(M, T, "RK   ", rk);
    run(M, T, "BITAP", bitap);
}
```



Experimental findings – Random Strings

Pattern	Time in microseconds				
m	Brute Force	KMP	BMH	RK	BITAP
2	32.81333	19.77967	33.70487	45.4758	27.41065
3	25.31113	11.50986	14.28385	40.6214	21.51133
4	25.52351	11.7321	10.53296	41.37416	21.68629
5	25.74828	11.56085	8.83097	40.95325	21.70076
6	26.31965	11.31291	7.48713	40.91029	22.04134
7	26.34072	11.60992	6.22689	40.90109	21.60537
8	25.52076	11.53245	7.10483	42.05945	21.9666
9	25.91869	11.27961	5.43424	41.07827	21.83601
10	25.7609	11.71329	4.50006	40.75521	21.6186

Table 1. Execution times of random strings

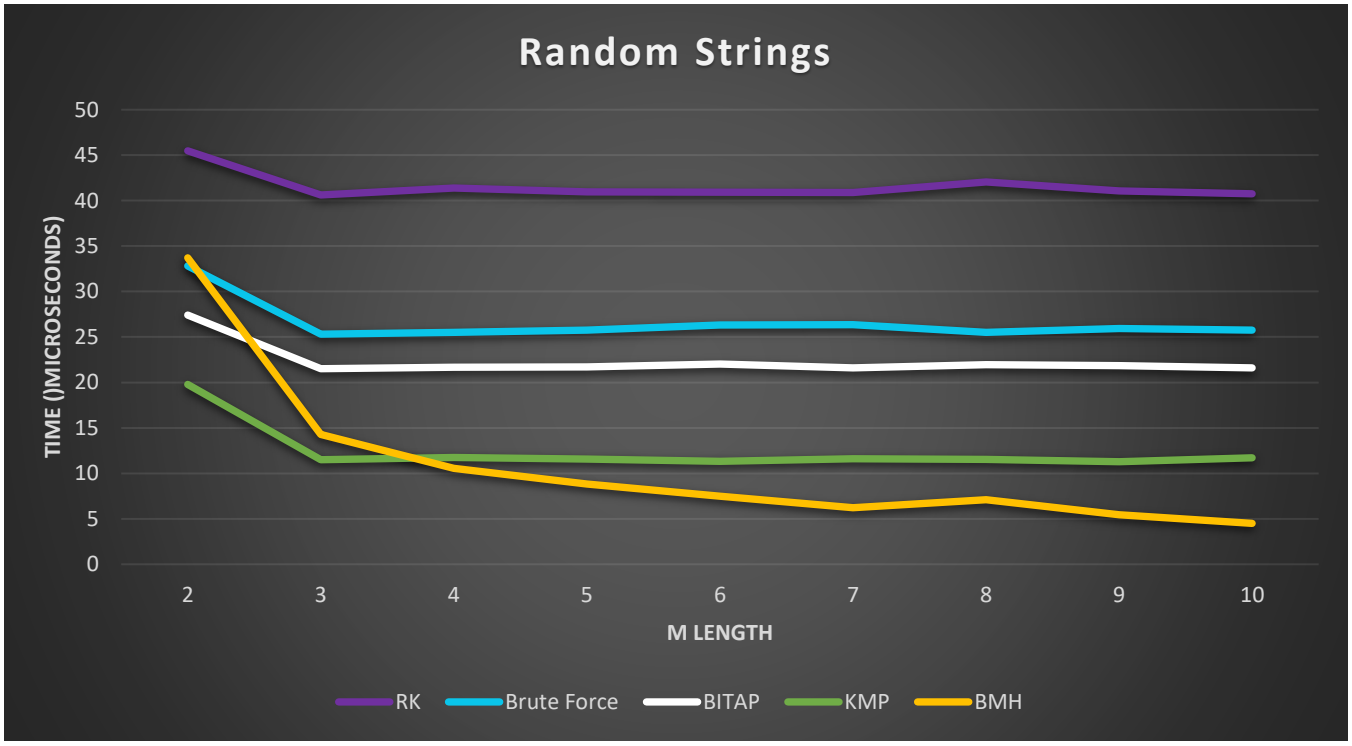


Fig 1. Plot of execution times of random strings

Experimental findings – English Words

Pattern	Time in microseconds				
m	Brute Force	KMP	BMH	RK	BITAP
2	79.57688	36.89431	78.64268	129.74005	97.50817
3	98.17098	62.53191	58.61704	162.82624	96.33735
4	110.33659	58.61114	50.0703	170.57134	101.65993
5	115.41107	55.16546	44.38198	178.45565	106.15033
6	120.03385	65.71231	39.0368	179.27862	107.12553
7	121.74034	83.18445	34.50343	180.43071	107.2616
8	123.32408	56.76527	31.17611	179.86031	107.34801
9	123.86736	58.00938	28.71236	180.05006	107.41769
10	125.4105	60.30199	26.78517	180.39783	107.59897

Table 2. Execution times of English words

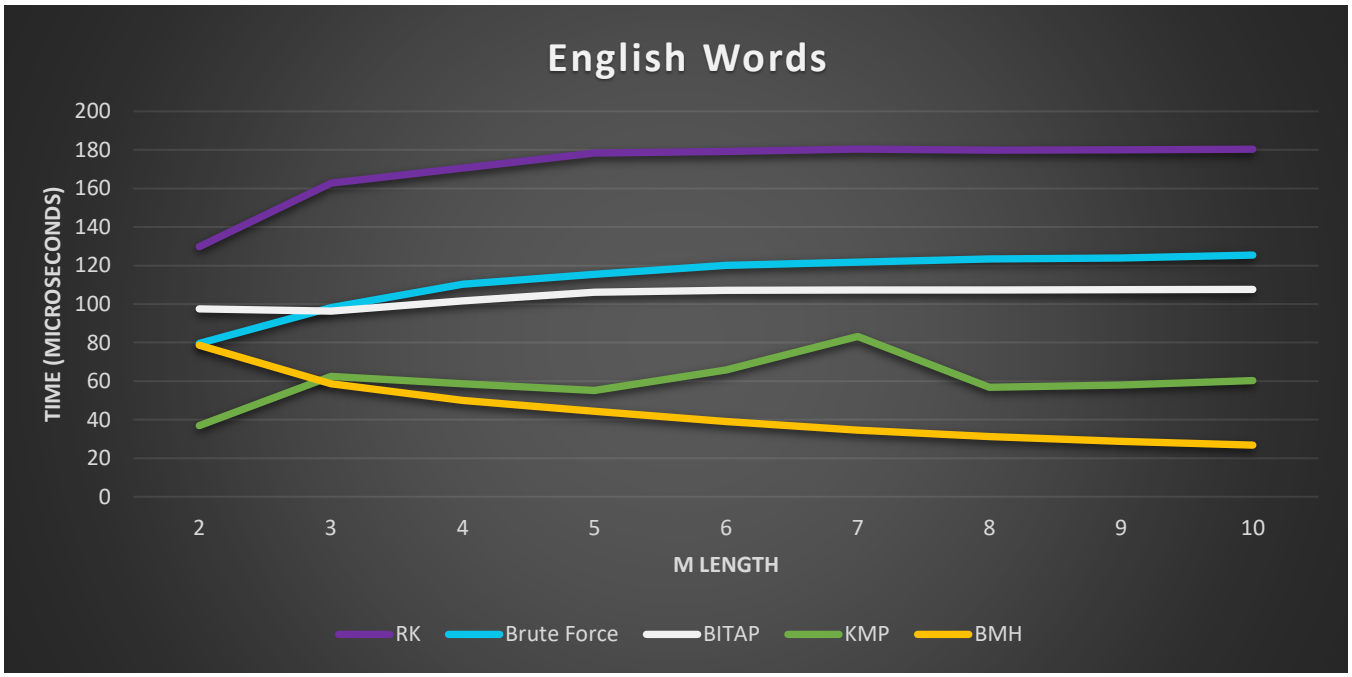


Fig 2. Plot of execution times of English words

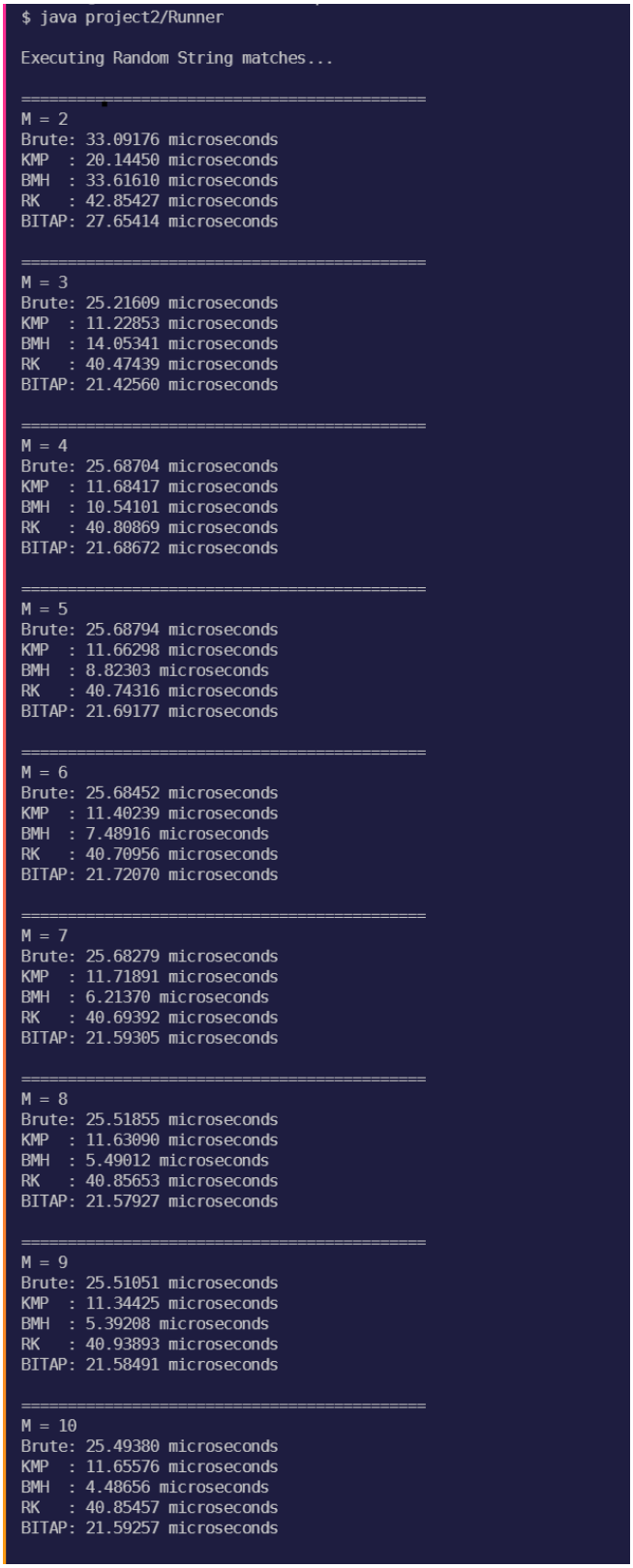
Conclusions and observations

- From the above plots, it is clear that the best algorithm for both random strings and English words is **Boyer-Moore-Harspool** . The **trendline for BMH** shows that it steadily decreases as the length of m increases. This is highly desirable — since in real-world, extremely small patterns are unlikely.
- Only for extremely small pattern length 1 or 2, it can be seen that BMH falls behind other algorithms. But for other lengths it is better than almost every other algorithm. This is due to the fact that the Last Occurrence Table would mostly be empty for the Alphabet size, and it essentially becomes sliding P across T. Therefore, it tries to match almost every single character of T with P.
- Furthermore, my empirical results show that **Rabin-Karp performs the worst**. This is surprising but understandable because even though the average and best-case complexity of Rabin-Karp is $\boldsymbol{O(m+n)}$, its worst-case is $\boldsymbol{O(mn)}$ and is inferior to other algorithms. Furthermore, it is noteworthy to point out that it performs worse than brute force — this could be due to the **overhead of Cyclic Polynomial Rolling Hash and Bit manipulation**. If we have multiple patterns to match, a modified Rabin-Karp algorithm might perform better.
- Both **KMP and Bitap** perform better than **brute force**. However, if our pattern size tends to be larger than the **word length** of the machine, KMP would significantly outperform Bitap. Bitap also prefers significantly smaller Alphabet size.
- Interestingly, even though theoretical asymptotic efficiencies hint that KMP would be the best performer, in real-world scenarios **constant factors add-up very quickly**. Linear worst-case algorithms such as KMP have a significant overhead of pre-processing the entire Pattern String. For this reason, **KMP is also not space optimal for large P**.
- Unlike KMP, BMH takes advantage of the fact that in real-word scenarios Alphabet size is considerably limited — hence preprocessing is faster for longer Pattern Strings. It is also not limited by the word length of the machine like Bitap. Making it the best algorithm for our scenario.

Algorithm	Preprocessing required?	Performs best when	Performs worst when
Brute	No	Pattern present early on	Pattern repeatedly present with character mismatch towards the end
Rabin-Karp	Yes	Multi pattern search	Hash collision/overhead and limited size of integer data type size
KMP	Yes	Pattern significantly smaller than alphabet size. Example: Large files	Large Pattern and large alphabet (more mismatches)
Bitap	Yes	Significantly small alphabet size	Alphabet size increases
Boyer-Moore-Harspool	Yes	Large pattern length and small alphabet size	Low bad character skip

Screenshots from *openlab.ics.uci.edu*

- Random Strings:



```
$ java project2/Runner

Executing Random String matches...

=========================================
M = 2
Brute: 33.09176 microseconds
KMP  : 20.14450 microseconds
BMH  : 33.61610 microseconds
RK   : 42.85427 microseconds
BITAP: 27.65414 microseconds

=========================================
M = 3
Brute: 25.21609 microseconds
KMP  : 11.22853 microseconds
BMH  : 14.05341 microseconds
RK   : 40.47439 microseconds
BITAP: 21.42560 microseconds

=========================================
M = 4
Brute: 25.68704 microseconds
KMP  : 11.68417 microseconds
BMH  : 10.54101 microseconds
RK   : 40.80869 microseconds
BITAP: 21.68672 microseconds

=========================================
M = 5
Brute: 25.68794 microseconds
KMP  : 11.66298 microseconds
BMH  : 8.82303 microseconds
RK   : 40.74316 microseconds
BITAP: 21.69177 microseconds

=========================================
M = 6
Brute: 25.68452 microseconds
KMP  : 11.40239 microseconds
BMH  : 7.48916 microseconds
RK   : 40.70956 microseconds
BITAP: 21.72070 microseconds

=========================================
M = 7
Brute: 25.68279 microseconds
KMP  : 11.71891 microseconds
BMH  : 6.21370 microseconds
RK   : 40.69392 microseconds
BITAP: 21.59305 microseconds

=========================================
M = 8
Brute: 25.51855 microseconds
KMP  : 11.63090 microseconds
BMH  : 5.49012 microseconds
RK   : 40.85653 microseconds
BITAP: 21.57927 microseconds

=========================================
M = 9
Brute: 25.51051 microseconds
KMP  : 11.34425 microseconds
BMH  : 5.39208 microseconds
RK   : 40.93893 microseconds
BITAP: 21.58491 microseconds

=========================================
M = 10
Brute: 25.49380 microseconds
KMP  : 11.65576 microseconds
BMH  : 4.48656 microseconds
RK   : 40.85457 microseconds
BITAP: 21.59257 microseconds
```

- English Words:

```
Executing English word matches...

==========================================
M = 2
Brute: 81.13885 microseconds
KMP  : 37.00685 microseconds
BMH  : 80.63475 microseconds
RK   : 135.39251 microseconds
BITAP: 96.25218 microseconds

==========================================
M = 3
Brute: 98.43857 microseconds
KMP  : 64.85531 microseconds
BMH  : 56.20978 microseconds
RK   : 160.97909 microseconds
BITAP: 95.31727 microseconds

==========================================
M = 4
Brute: 113.18915 microseconds
KMP  : 45.33125 microseconds
BMH  : 49.27327 microseconds
RK   : 168.77698 microseconds
BITAP: 100.52441 microseconds

==========================================
M = 5
Brute: 116.94712 microseconds
KMP  : 48.84299 microseconds
BMH  : 44.74250 microseconds
RK   : 178.29368 microseconds
BITAP: 107.10072 microseconds

==========================================
M = 6
Brute: 119.84806 microseconds
KMP  : 52.10202 microseconds
BMH  : 39.06986 microseconds
RK   : 178.79147 microseconds
BITAP: 107.01682 microseconds

==========================================
M = 7
Brute: 121.78047 microseconds
KMP  : 77.95015 microseconds
BMH  : 35.05941 microseconds
RK   : 180.23182 microseconds
BITAP: 107.29925 microseconds

==========================================
M = 8
Brute: 123.66076 microseconds
KMP  : 62.67563 microseconds
BMH  : 31.58803 microseconds
RK   : 181.33488 microseconds
BITAP: 107.89859 microseconds

==========================================
M = 9
Brute: 123.88495 microseconds
KMP  : 58.45241 microseconds
BMH  : 28.70399 microseconds
RK   : 180.34961 microseconds
BITAP: 107.60400 microseconds

==========================================
M = 10
Brute: 125.67780 microseconds
KMP  : 60.94490 microseconds
BMH  : 26.50869 microseconds
RK   : 180.04053 microseconds
BITAP: 107.54691 microseconds
```