A quantitative approach to Pattern search algorithm optimization

**Introduction** : Nowadays data is growing tremendously. Therefore, there is a great need to store and process data. The problem of Pattern Searching has different applications . The purpose of Pattern searching is to find occurrence of a given pattern within another text. Algorithms like KMP, Rabin -karp , boyer -Moore , are some essential manifestation of pattern searching technique. It finds its application in , text compression , information retrieval, DNA matching .etc

**Problem**: find all the occurrence of a pattern in a database. a pattern represents a non-empty language that contains symbol other than the empty string. The problem of pattern searching is to search for occurrences of strings of language in other strings – or in texts that are less formal. A pattern p = {δ}n , δ ∈ Σ ( input symbols set ) |p| > 0 i.e n > 0 database size Q ={δ}m  |Q| > 0 i.e m >0 and m ≥ n

**Background:**  • Space Complexity – which actually represents the necessary memory or space required by the algorithm to correctly execute the inputs.

• Time Complexity – which actually represents the time required for the algorithm to  
correctly execute the inputs .

We’ ll be performing Algorithm analysis i.e the estimation of the resources needed for an algorithm to solve a given problem. To be exact we will be focusing more on temporal and spatial complexity .





|  |  |  |  |
| --- | --- | --- | --- |
| Naive string search algorithm | O(1) | O(1) | O((n-m+1) m) |
| Rabin–Karp string search algorithm | O(m) | O(1) | Average O(n+m), worst O((n-m+1) m) |
| Knuth–Morris–Pratt algorithm | O(m) | O(m) | O(m+n) |
| Boyer–Moore string search algorithm | O(m + |Σ|) | O(m + |Σ|) | Ω(n/m), O(n) |

**Simulation:**

**Rabin- Karp**

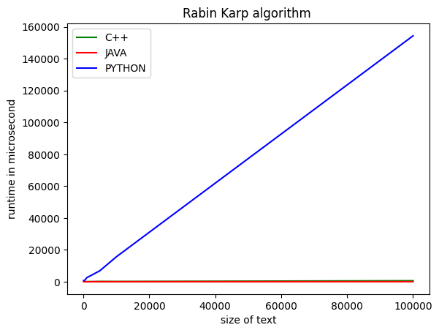
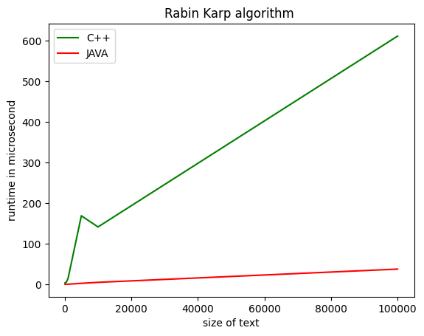
|  |  |  |  |
| --- | --- | --- | --- |
| Text size | C++ | Java | Python |
| 100 | 3.6 | 0.2 | 777 |
| 500 | 5.5 | 0.21 | 1011 |
| 1000 | 14.3 | 0.32 | 2477 |
| 5000 | 168.8 | 2.5 | 6882 |
| 10000 | 141.3 | 4.9 | 15627 |
| 100000 | 611 | 37.5 | 154368 |

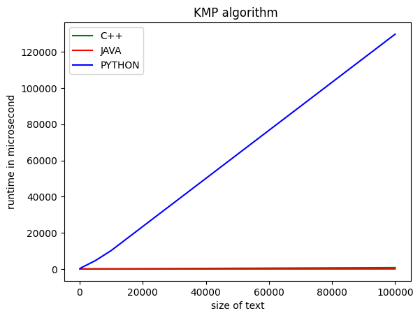
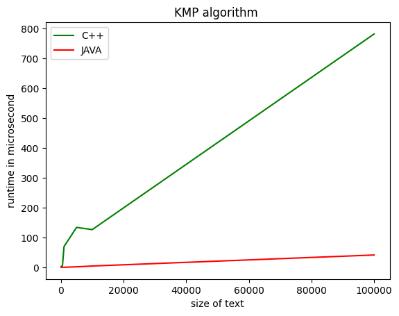
**knuth morris pratt**

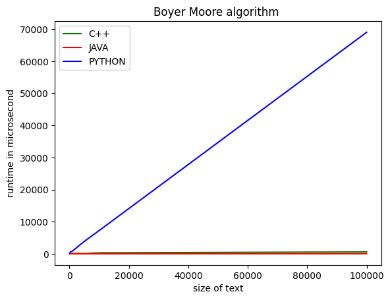
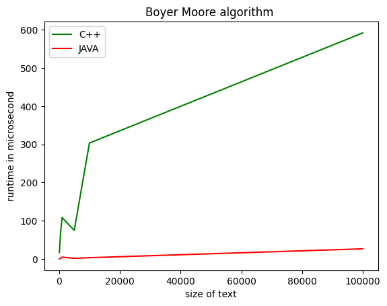
|  |  |  |  |
| --- | --- | --- | --- |
| Text size | C++ | java | python |
| 100 | 2.8 | 0 | 199.9 |
| 500 | 5.2 | 0.1 | 723.5 |
| 1000 | 69.5 | 0.3 | 1148.3 |
| 5000 | 133.8 | 1.8 | 4674.1 |
| 10000 | 126.3 | 4.4 | 10216.4 |
| 100000 | 781.4 | 41.3 | 129764.1 |

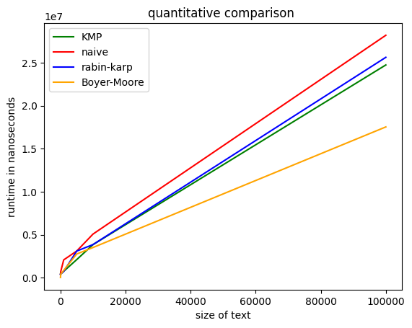
**Boyer – Moore**

|  |  |  |  |
| --- | --- | --- | --- |
| Text size | C++ | java | python |
| 100 | 16.6 | 0 | 199 |
| 500 | 69.4 | 1.2 | 659 |
| 1000 | 108.2 | 4.7 | 811 |
| 5000 | 74.6 | 1.4 | 3848 |
| 10000 | 303.4 | 3.1 | 7222 |
| 100000 | 591.8 | 26.3 | 69011 |

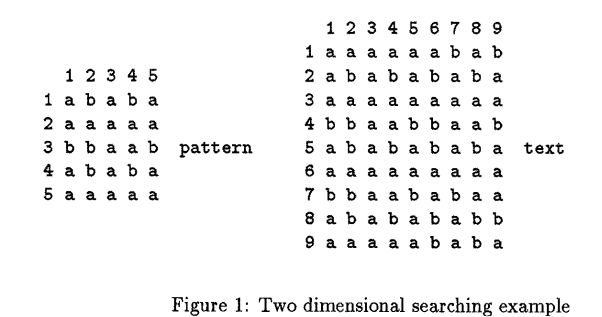
 



The above runtime simulation helps us visualize efficiency of the algorithms.

Some important insight -

* As the size of text increases efficiency is more pronounced
* but at relatively smaller size of text there are discrepancies , this may be attributed to spurious hits in Rabin Karp , and space complexity of KMP.
* Spurious hit can be reduced by optimising the hash function
* For multi patterns searching KMP has to be applied multiple times. So, it is not feasible in case of multiple patterns or texts i.e only 1D matching is feasible. In that case, more advanced data structures like: Trie, Suffix Trees or Suffix arrays are used.
* Rabin karp is efficient in resolving the above issue i.e multidimensional pattern matching .



* Boyer moore significantly outperforms KMP
* Together KMP and rabin karp could be merged to reach a new algorithm when the number of types of the pattern is relatively big, or the number is small but unevenly distributed the merged form outperforms the individualistic algo.

