**TIME COMPLEXITY AND APPLICATIONS OF STRING SEARCH ALGORITHIMS**

**Brute force:**

Brute force pattern matching runs in time O(mn) in the worst case.Average for most searches of ordinary text take O(m+n), which is very quick. Example of a more average case: T: "a string searching example is standard" P: "store".

**Brute force Application:**

a brute-force search is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases. Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific [heuristics](https://en.wikipedia.org/wiki/Heuristic_(computer_science)) that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

**Rabin Karp:**

The **average** and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the **hash values** of all the substrings of txt[] match with **hash value** of pat[].

**Rabin Karp Application:**

A practical application of the algorithm is [detecting plagiarism](https://en.wikipedia.org/wiki/Plagiarism_detection). Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical.

This algorithm works well in many practical cases, but can exhibit relatively long running times on certain examples, such as searching for a pattern string of 10,000 "a"s followed by a single "b" in a search string of 10 million "a"s, in which case it exhibits its worst-case [O](https://en.wikipedia.org/wiki/Big-O_notation)(*mn*) time.

**Boyer Moore**

The Bad Character Heuristic may take O(mn)time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA.

**Boyer Moore Application:**

When the pattern *does* occur in the text, running time of the original algorithm is

O ( n m ) in the worst case. This is easy to see when both pattern and text consist solely of the same repeated character. . In [C++](https://en.wikipedia.org/wiki/C%2B%2B), [Boost](https://en.wikipedia.org/wiki/Boost_(C%2B%2B_libraries)) provides the [generic Boyer–Moore search](http://www.boost.org/doc/libs/1_58_0/libs/algorithm/doc/html/algorithm/Searching.html#the_boost_algorithm_library.Searching.BoyerMoore) implementation under the *Algorithm* libraryIn general, the algorithm runs faster as the pattern length increases.

**Naïve:**

The best case occurs when the first character of the pattern is not present in text at all.

|  |
| --- |
| txt[] = "AABCCAADDEE";  pat[] = "FAA"; |

The number of comparisons in best case is O(n).The worst case of Naive Pattern Searching occurs in following scenarios.  
1) When all characters of the text and pattern are same.

|  |
| --- |
| txt[] = "AAAAAAAAAAAAAAAAAA";  pat[] = "AAAAA"; |

2) Worst case also occurs when only the last character is different.

|  |
| --- |
| txt[] = "AAAAAAAAAAAAAAAAAB";  pat[] = "AAAAB"; |

Number of comparisons in worst case is O(m\*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to O(n).

Let *m* be the length of the pattern, *n* be the length of the searchable text and *k* = |Σ| be the size of the alphabet.

**Naïve Application:**

For most practical purposes, which deal with texts based on human languages, this approach is much faster since the inner loop usually quickly finds a mismatch and breaks. A problem arises when we are faced with different kinds of “texts,” such as the genetic code.

**Finite Automata:**

If we reach the final state, then the pattern is found in the text. The time complexity of the search process is O(n).

**Finite Automata Application:**

Consider ﬁnding all occurrences of a short string (pattern string) within a long string (text string). This can be done by processing the text through a DFA: the DFA for all strings that end with the pattern string. Each time the accept state is reached, the current position in the text is output.

Goddard

**Anagram:**

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] and its permutations (or anagrams) in txt[]. You may assume that n > m.

Expected time complexity is O(n)

**Anagram Application:**

Expected time complexity is O(n)

This problem is slightly different from standard pattern searching problem, here we need to search for anagrams as well.

We can achieve O(n) time complexity under the assumption that alphabet size is fixed which is typically true as we have maximum 256 possible characters in ASCII. The idea is to use two count arrays:

1) The first count array store frequencies of characters in pattern.

2) The second count array stores frequencies of characters in current window of text.

The important thing to note is, time complexity to compare two count arrays is O(1) as the number of elements in them are fixed (independent of pattern and text sizes).

**KMP:**

Since the two portions of the algorithm have, respectively, complexities of O(k) and O(n), the complexity of the overall algorithm is O(n + k). So the total cost of a KMP search is **linear** in the number of characters of string and pattern.

**KMP Application:**

In real world **KMP** algorithm is used in those **applications** where pattern matching is done in long strings, whose symbols are taken from an alphabet with little cardinality.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Preprocessing time** | **Matching time**[[1]](https://en.wikipedia.org/wiki/String_searching_algorithm#endnote_Asymptotic_times) | **Space** |
| **Naïve string search algorithm** | none | Θ(nm) | none |
| [**Rabin–Karp string search algorithm**](https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_string_search_algorithm) | Θ(m) | average Θ(n + m), worst Θ((n−m)m) | O(1) |
| [**Knuth–Morris–Pratt algorithm**](https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm) | Θ(m) | Θ(n) | Θ(m) |
| [**Boyer–Moore string search algorithm**](https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm) | Θ(m + k) | best Ω(n/m), worst O(mn) | Θ(k) |
| [**Bitap algorithm**](https://en.wikipedia.org/wiki/Bitap_algorithm) **(*shift-or*, *shift-and*, *Baeza–Yates–Gonnet*)** | Θ(m + k) | O(mn) |  |
| [**Two-way string-matching algorithm**](https://en.wikipedia.org/w/index.php?title=Two-way_string-matching_algorithm&action=edit&redlink=1) | Θ(m) | O(n+m) | O(1) |
| **BNDM (Backward Non-Deterministic Dawg Matching)** | O(m) | O(n) |  |
| **BOM (Backward Oracle Matching)** | O(m) | O(n) |  |