Exact Pattern Matching

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Reuse information after a mismatch

Exact pattern matching asks where a short pattern occurs in a longer text. The naive method forgets every comparison after a mismatch. Faster methods retain different kinds of evidence: a matched border, a rolling fingerprint, a suffix of the current window, a word of automaton states, or a trie prefix shared by many patterns.

No one method dominates every workload. Pattern length, alphabet, number of patterns, preprocessing budget, and the need for worst-case guarantees all matter.

2 Learning goals

After studying this note, you should be able to:

- state exact-search inputs and output conventions precisely;
- trace naive search and KMP, including its linear-time prefix data;
- derive rolling hashes and Boyer–Moore-style shifts;
- trace Shift-And and backward-oracle search; and
- build and search an Aho–Corasick automaton for multiple patterns.

3 Prerequisites

You should know arrays, modular arithmetic, hashing, tries, and asymptotic notation. The tree note helps with tries; the following automata note supplies general theory but is not required.

4 Problem model and conventions

Let text $T[0..n)$ and nonempty pattern $P[0..m)$ be strings over alphabet Σ, with $m \le n$. An occurrence at shift s satisfies

$$T[s..s+m) = P[0..m).$$

Unless stated otherwise, an algorithm reports **all** shifts, including overlapping occurrences. Reporting z positions costs $\Omega(z)$ time. Empty-pattern semantics differ between libraries and are outside this model.

State the alphabet-lookup model (constant-time array, expected-time hash, or ordered map), separate preprocessing from scanning, and distinguish character comparisons from word operations.

5 Baseline: try every alignment

```
naive(T, P):
    for s = 0 .. n-m:
        j = 0
        while j < m and T[s+j] = P[j]:
            j = j+1
        if j = m: report s
```

The loop checks every possible alignment and reports it exactly when all m characters agree. Its worst-case time is $\Theta((n-m+1)m)$, for example on repeated letters. It is still attractive for tiny patterns and as a correctness reference.

The central question is: after a mismatch, which facts about the already-read text remain useful?

6 Knuth–Morris–Pratt: preserve a border

A **border** of a string is a proper prefix that is also a suffix. Define $\pi[i]$ as the length of the longest border of $P[0..i]$. The prefix function is itself computed by border fallback:

```
prefix_function(P):
    pi[0] = 0; q = 0
    for i = 1 .. m-1:
        while q > 0 and P[q] != P[i]: q = pi[q-1]
        if P[q] = P[i]: q = q+1
        pi[i] = q
    return pi
```

During search, q is the length of the longest pattern prefix that is a suffix of the text read so far:

```
kmp(T, P):
    pi = prefix_function(P)
    q = 0
    for i = 0 .. n-1:
        while q > 0 and P[q] != T[i]:
            q = pi[q-1]
        if P[q] = T[i]: q = q+1
        if q = m:
            report i-m+1
            q = pi[m-1]       # retain a border; allow overlaps
```

The invariant explains correctness. If the next characters mismatch, every alignment with a longer retained prefix has already been contradicted. The longest possible remaining candidate is a border, and repeated fallback enumerates shorter borders without moving backward in the text.

7 Worked trace: KMP fallback

For `ABABAC`, the prefix values are `0 0 1 2 3 0`. In text `ABABABAC`, five characters match at shift 0 before `B` mismatches expected `C`. KMP falls from $q = 5$ to $\pi[4] = 3$, reuses the known suffix `ABA`, matches the same `B`, and eventually reports shift 2.

Every successful comparison increases q and advances the text. Every fallback decreases q, and q cannot decrease more in total than it has increased. Prefix construction is $O(m)$, scanning is $O(n)$, and stored prefix data is $O(m)$.

8 Rabin–Karp: compare fingerprints

Encode symbols as integers, choose base b and modulus q, and define the length-m fingerprint

$$H(S) = \sum_{j=0}^{m-1} code(S[j]) b^{m-1-j} \bmod q.$$

Let $h = b^{m-1} \bmod q$. If $W_s = T[s..s+m)$, the next window hash is

$$H(W_{s+1}) = ((H(W_s) - code(T[s])h)b + code(T[s+m])) \bmod q.$$

Thus each shift takes constant modular arithmetic. When the pattern and window hashes agree, compare their characters before reporting. In an implementation, normalise a negative remainder after subtracting the outgoing character. Verification makes the algorithm exact; a collision only adds work.

With a suitably random modulus or universal fingerprint family, expected **false-collision** work is small. The rolling scan itself is $O(n + m)$, but an exact all-occurrences implementation that compares all m characters for every equal fingerprint also spends $O(zm)$ on its z true matches. Its expected bound is therefore $O(n + m + zm)$ under a hash choice that keeps expected false-collision verification linear.

This expectation is about the hash choice, not vaguely about "ordinary text." Repetitive matches, or an adversarial or unlucky sequence of collisions, can produce $O(nm)$ verification time. Multiple equal-length patterns can share the same rolling scan by mapping fingerprints to candidate patterns.

9 Boyer–Moore family: inspect from the right

Boyer–Moore aligns a pattern window but compares from right to left. A mismatch may prove that several following alignments are impossible.

For a mismatch at pattern position j against text character c, let $last_{<j}(c)$ be the rightmost occurrence of c in $P[0..j)$, or -1 if absent. The bad-character shift is

$$\max(1, j - last_{<j}(c)).$$

The **good-suffix rule** uses the suffix already matched to align another occurrence of that suffix, or its longest suffix that is also a pattern prefix. A correct implementation takes the maximum safe shift supplied by its rules.

Horspool is simpler: after an attempt, it shifts according to the text character currently under the pattern's last position. It often performs very few comparisons on long patterns over moderate alphabets, but its worst case is $O(nm)$. Basic Boyer–Moore variants also differ in worst-case guarantees; linear worst-case claims require the exact preprocessing and matching variant to be named.

10 Shift-And: simulate prefixes in one word

Let bit j mean that prefix $P[0..j]$ matches a suffix of the text processed so far. For each character c, mask $M[c]$ has bit j set exactly when $P[j] = c$. Starting with $D = 0$, read character c using

$$D = ((D \ll 1) \mid 1) \,\&\, M[c].$$

If bit $m-1$ is set, an occurrence ends at the current position. The shift extends every active prefix, the inserted low bit begins a new attempt, and the mask retains only character-compatible states.

If $m \leq w$ for machine-word size w, scanning takes $O(n)$ word operations and $O(|\Sigma|)$ masks of w bits. Longer patterns require $\lceil m/w \rceil$ words and $O(n\lceil m/w \rceil)$ word operations. A Shift-Or implementation uses complemented bits; mixing the two conventions is a common source of errors.

11 Factor oracles and backward matching

A **factor** is a contiguous substring. Right-to-left **factor methods** test a suffix of each text window against an automaton representing factors of the pattern. A factor oracle for word $X[1..m]$ is an acyclic automaton with states $0..m$, at most $2m-1$ transitions, and a supply link $S[i]$ from each noninitial state. It accepts every factor of X and may accept extra strings; those false positives affect shifts, not final exact verification.

It can be built online:

```
factor_oracle(X):
    S[0] = -1
    for i = 1 .. m:
        add transition (i-1) --X[i]--> i
        p = S[i-1]
        while p != -1 and transition(p, X[i]) is undefined:
            add transition p --X[i]--> i
            p = S[p]
        if p = -1: S[i] = 0
        else:      S[i] = transition(p, X[i])
```

Backward Oracle Matching (BOM) builds the oracle of the reversed pattern. At window start s, it feeds $T[s+m-1], T[s+m-2], ...$ to the oracle. If the transition for $T[s+i]$ fails after reading the suffix to its right, then that failed character followed by the read suffix is not a factor of P. The window may safely move past that character:

```
bom(T, P):
    O = factor_oracle(reverse(P))
    s = 0
    while s <= n-m:
        state = 0
        i = m-1
        while i >= 0 and transition(state, T[s+i]) is defined:
            state = transition(state, T[s+i])
            i = i-1
        if i < 0: report s
        if i < 0: s = s+1
        else:     s = s+i+1
```

The full backward read still verifies every reported match. Construction is $O(m)$ with suitable transition maps; the simple search has $O(nm)$ worst-case time but can skip much of typical text. BDM uses an exact suffix automaton; BOM saves structure by using the permissive oracle.

12 Multiple patterns: Aho–Corasick

A trie shares prefixes of a dictionary. Aho–Corasick adds a **failure link** from a trie node to the longest proper suffix of its path label that is also a trie prefix. A terminal node stores the identifiers of patterns ending there. An **output link** can point to the nearest terminal node on its failure chain, so suffix-pattern outputs need not be copied into every node.

Insert every pattern into the trie. Then process nodes in breadth-first order. Root children fail to the root. For every trie edge $r \xrightarrow{a} s$, follow failure links from $fail[r]$ until an a transition is available or the root is reached; that destination becomes $fail[s]$. Set s's output link to that destination if it is terminal, or otherwise to the destination's output link.

During scanning, follow an a edge if possible; otherwise follow failures until one becomes possible or the root is reached. Report the terminal identifiers at the current state and along its output-link chain. For patterns `he`, `she`, `his`, and `hers`, reaching the state for `she` reports both `she` and `he` because the `she` state's output link reaches `he`.

Operationally, an output should contain at least a pattern identifier and the one-past-the-last text boundary `end`. Its start is `end - pattern_length`; storing only the matched text does not distinguish duplicate dictionary entries or recover their identifiers.

If the dictionary has total length L, construction with output links is $O(L)$ plus alphabet-transition costs. Scanning is $O(n + z)$ with constant-time transitions: each failure shortens the current trie depth, so total failure work is amortised linear, and traversed output links account for reported matches. Dense transition tables cost $O(|Q||\Sigma|)$ space; sparse maps save space with a lookup-time trade-off. Physically copying inherited output lists is also correct, but its preprocessing time and storage include the number of copied identifiers and need not remain $O(L)$.

Commentz–Walter combines a backward trie with Boyer–Moore-style shifts. Wu–Manber hashes short blocks near the window end to keep useful shifts when there are many patterns. They can outperform Aho–Corasick on suitable data, but their behaviour depends on shortest-pattern length, alphabet, and distribution; Aho–Corasick retains the clearest worst-case output-sensitive guarantee.

13 Complexity and method trade-offs

Method	Preprocessing	Worst-case scan	Main condition or strength
Naive	$O(1)$	$O(nm)$	tiny patterns; reference implementation

Method	Preprocessing	Worst-case scan	Main condition or strength
KMP	$O(m)$	$O(n)$	deterministic one-pattern guarantee
Rabin–Karp	$O(m)$	$O(nm)$	rolling scan; good expected hashing behaviour
Horspool	$O(m + \|\Sigma\|)$	$O(nm)$	often large practical shifts
Shift-And	masks	$O(n\lceil m/w \rceil)$ word operations	short patterns and bit operations
BOM	$O(m)$	$O(nm)$	factor-based backward skips
Aho–Corasick	dictionary length	$O(n + z)$	many patterns with output guarantee

14 Connections

- KMP failure and Aho–Corasick failure links preserve the longest useful suffix.
- Shift-And is a finite-automaton simulation packed into bits.
- The next note formalises finite automata and regular-expression compilation.
- Full-text indexes later preprocess the text instead of the pattern and support many future queries.

15 Common mistakes

- Reporting hash equality without character verification.
- Computing KMP fallback from text rather than pattern borders.
- Forgetting the post-match KMP fallback needed for overlaps.
- Using a bad-character occurrence to the right of the mismatch.
- Claiming every Boyer–Moore variant has the same guarantee.
- Hiding the word-size limit of a bit-parallel method.
- Assuming a factor oracle recognises exactly, rather than at least, the pattern factors.
- Ignoring z, the number of reported matches.

16 Self-check

1. Compute π for `ABABACA` and explain every nonzero value.
2. Why does KMP never need to move the text index backward?
3. Derive one Rabin–Karp rolling update by expanding both polynomial hashes.
4. Compute a bad-character shift when `c` mismatches pattern position j and occurs only after j.
5. Trace four Shift-And updates for pattern `aba`.
6. What property makes the simplified BOM shift after a failed character safe?
7. Why can one Aho–Corasick state report several patterns?

17 Revision summary

- Exact matchers differ mainly in the evidence they retain after mismatch.
- KMP uses borders and gives deterministic linear scanning.
- Rabin–Karp uses rolling fingerprints but remains exact through verification.
- Boyer–Moore and factor methods read windows backward to obtain safe skips.
- Bit parallelism updates many prefix states in each machine operation.
- Aho–Corasick extends trie prefixes and failure links to many patterns.

18 Implementations and hands-on exploration

- **Python:** `str.find` and `bytes.find` return the lowest matching index from an optional start position. Repeated calls are needed for all occurrences, and advancing by one rather than by pattern length is required to retain overlaps. A `str` index counts Unicode code points in Python's string representation, whereas a `bytes` index counts bytes; neither is a grapheme-cluster index.
- **C++:** `std::boyer_moore_searcher` is a standard-library searcher usable with `std::search`. It needs random-access pattern and text ranges, and its hash and equality predicate must agree on which symbols are equal.
- **Rust:** `aho-corasick` implements multi-pattern automata with noncontiguous-NFA, contiguous-NFA, and DFA trade-offs. Only standard match semantics support the overlapping iterator; leftmost-first and leftmost-longest intentionally suppress some overlaps. Match spans are half-open byte offsets, even when patterns and text are supplied as UTF-8 strings.
- **Java:** `String.indexOf` returns the first substring occurrence and can resume from a supplied index. Java indices count UTF-16 code units, so a supplementary Unicode character occupies two positions.

18.1 Small experiments

1. Find every occurrence of `aa` in `aaaa` with repeated Python `find` calls and with Rust `aho-corasick`. First request non-overlapping matches, then all overlaps, and explain the different position lists.
2. Implement the naive matcher and KMP, then compare both against a library search on random text and on `a^n`. Assert identical occurrence lists and measure preprocessing separately from scanning.

19 Sources and further study

- Exact matching lecture deck, Jaak Vilo.
- Backward Oracle Matching algorithm description, Thierry Lecroq.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 4th ed., string-matching material.