Boyer-Moore

Can we improve on the naïve algorithm?

```
P: word
T: There would have been a time for such a word
         word
```

u doesn't occur in *P*, so skip next two alignments

```
P: word
T: There would have been a time for such a word
         word
          word    skip!
           word    skip!
            word
```

Boyer-Moore

Learn from character comparisons to skip pointless alignments

1. When we hit a mismatch, move *P* along until the mismatch becomes a match — "Bad character rule"
2. When we move *P* along, make sure characters that matched in the last alignment also match in the next alignment — "Good suffix rule"
3. Try alignments in one direction, but do character comparisons in *opposite* direction — For longer skips

P: **`word`**

T: **`There would have been a time for such a word`**

`         word`

Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character.

(c) If there was no mismatch, don't skip

Step 1:
T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
P: C C T T T T G C
Case (a)

Step 2:
T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
P: C C T T T T G C
Case (b)

Step 3:
T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
P: C C T T T T G C
Case (c)

Step 4:
T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
P: C C T T T T G C

(etc)

Boyer-Moore: Bad character rule

```
Step 1:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P: C C T T T T G C

Step 2:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P:       C C T T T T G C

Step 3:  T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
         P:                     C C T T T T G C
```

Up to step 3, we skipped 8 alignments

5 characters in *T* were never looked at

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t* *or* (b) *P* moves past *t*

```
                     t
                   |---|
Step 1:   T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
          P: C T T A C T T A C

                     t
                   |-----------|
Step 2:   T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
          P:         C T T A C T T A C

Step 3:   T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
          P:                 C T T A C T T A C
```

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t* *or* (b) *P* moves past *t*

Step 1:

t

T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A

P: C T T A C T T A C

t occurs *in its entirety* to the left within *P*

Step 2:

t

T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A

P: C T T A C T T A C

prefix of *P* matches a *suffix* of *t*

Step 3:

T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A

P: C T T A C T T A C

Case (a) has two subcases according to whether *t* occurs *in its entirety* to the left within *P* (as in step 1), or a *prefix* of *P* matches a *suffix* of *t* (as in step 2)

Boyer-Moore: Putting it together

How to *combine* bad character and good suffix rules?

```
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P:                     G T A G C G G C G
```

bad char says skip 2, good suffix says skip 7

Take the maximum! (7)

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

```
Step 1:  T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
         P: G T A G C G G C G                         bc: 6, gs: 0   bad character

Step 2:  T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
         P:               G T A G C G G C G           bc: 0, gs: 2   good suffix

Step 3:  T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
         P:                   G T A G C G G C G       bc: 2, gs: 7   good suffix

Step 4:  T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
         P:                                   G T A G C G G C G
```

11 characters of *T* we ignored

Step 1:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Step 2:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Step 3:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Step 4:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Skipped 15 alignments

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

P

Σ	T	C	G	C
A				
C		-		-
G			-	
T	-			

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

P

Σ

	T	C	G	C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2

T: **A A T C A A T A G C**
P: **T C G C**

This can be constructed efficiently. See Gusfield 2.2.2.

Boyer-Moore: Preprocessing

As with bad character rule, good suffix rule skips can be precalculated efficiently. See Gusfield 2.2.4 and 2.2.5.

For both tables, the calculations only consider *P*. No knowledge of *T* is required.

We'll return to preprocessing soon!

Boyer-Moore: Good suffix rule

We learned the *weak* good suffix rule; there is also a *strong* good suffix rule

```
                         t
                     |-------|
T:  C T T G C C T A C T T A C T T A C T
P:  C T T A C T T A C
   Weak:    C T T A C T T A C
              Strong: C T T A C T T A C
guaranteed
mismatch!
```

Strong good suffix rule skips more than weak, at no additional penalty

Strong rule is needed for proof of Boyer-Moore's O(n + m) worst-case time. Gusfield discusses proof(s) in first several sections of ch. 3

Boyer-Moore: Worst case

Boyer-Moore, with refinements in Gusfield, is O($n + m$) time

Given $n < m$, can simplify to O(m)

Is this better than naïve?

For naïve, worst-case # char comparisons is $n(m - n + 1)$

Boyer-Moore: O(m), naïve: O(nm)

Reminder: $|P| = n \quad |T| = m$

Boyer-Moore Algorithm

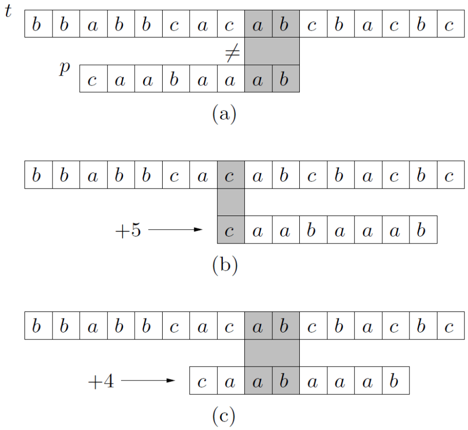


Figure: Boyer-Moore algorithm example

Bad Character Rule

- Pattern can be shifted to the rightmost occurrence of the symbol t_{j+i} in the pattern.
- Can propose a shift to the left but it will not be executed due to the good suffix rule.

Bad Character Rule

- Pattern can be shifted to the rightmost occurrence of the symbol t_{j+i} in the pattern.
- Can propose a shift to the left but it will not be executed due to the good suffix rule.
- In preprocessing, compute a function β that assigns the position of its last occurrence in p to each symbol from Σ (or the value 0, if the symbol does not occur in p).

Bad Character Rule

Algorithm 4.4 Preprocessing for the bad character rule

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

for all $a \in \Sigma$ **do** $\beta(a) := 0$
for $i := 1$ **to** m **do** $\beta(p_i) := i$

Output: The function β.

Figure: Preprocessing for the bad character rule

Bad Character Rule

Algorithm 4.4 Preprocessing for the bad character rule

Input: A pattern $p = p_1 \ldots p_m$ over an alphabet Σ.

for all $a \in \Sigma$ **do** $\beta(a) := 0$
for $i := 1$ **to** m **do** $\beta(p_i) := i$

Output: The function β.

Figure: Preprocessing for the bad character rule

- Running time: $O(m + |\Sigma|)$

Good Suffix Rule

- Pattern can be shifted to the next occurrence of the suffix $p_{i+1} \dots p_m$ in p.

Good Suffix Rule

- Pattern can be shifted to the next occurrence of the suffix $p_{i+1} \dots p_m$ in p.

Definition

Let s and t be two strings. We say that s is suffix similar to t, or $s \sim t$, if s is a suffix of t or t is a suffix of s.

Good Suffix Rule

- Pattern can be shifted to the next occurrence of the suffix $p_{i+1} \dots p_m$ in p.

Definition

Let s and t be two strings. We say that s is suffix similar to t, or $s \sim t$, if s is a suffix of t or t is a suffix of s.

- Good suffix rule: If $p_{i+1} \dots p_m = t_{j+i+1} \dots t_{j+m}$ and $p_i \neq t_{j+i}$, then shift the pattern for the next comparison to the right by $m - \gamma(i+1)$ positions, where

$$\gamma(i) = \max \{0 \leq k < m | p_i \dots p_m \sim p_1 \dots p_k\} .$$

Good Suffix Rule

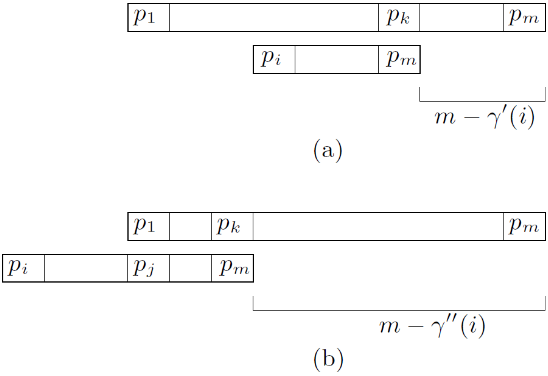


Figure: Good suffix rule cases

Good Suffix Rule

Algorithm 4.5 Preprocessing for the good suffix rule

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

1. Construct the string matching automaton $M = (Q, \Sigma, q_0, \delta, F)$ for $p_m \dots p_2$.
2. Determine the sequence $q_0, q_{m-1}, \dots, q_1$ of states M is traversing while reading the input $p_{m-1} \dots p_1$.
3. Compute the function γ':

   for $i := 2$ **to** m **do** $\gamma'(i) := 0$
   for $j := 1$ **to** $m - 1$ **do** $\gamma'(q_j) := j$
4. Compute the function γ'':

   for $i := 2$ **to** m **do**
     if $i \leqslant q_1$ **then**
       $\gamma''(i) := m - q_1 + 1$
     else
       $\gamma''(i) := 0$
5. Compute the function γ:

   for $i := 2$ **to** m **do** $\gamma(i) := \max\{\gamma'(i), \gamma''(i)\}$

Output: The function γ.

Figure: Good suffix rule algorithm

Finite Automaton

Definition

A finite automaton is a quintuple $M = (Q, \Sigma, q_0, \delta, F)$, where

- Q is a finite set of states,
- Σ is an input alphabet,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is a set of accepting states, and
- $\delta : Q \times \Sigma \to Q$ is a transition function describing the transitions of the automaton from one state to another.

We define the extension $\hat{\delta}$ of the transition function to strings over Σ by $\hat{\delta}(q, \lambda) = q$ and $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ for all $q \in Q$, $a \in \Sigma$ and $x \in \Sigma^*$. Thus, $\hat{\delta}(q, x)$ is the state M reaches from the state q by reading x.

We say that the automaton M accepts the string $x \in \Sigma^*$, if $\hat{\delta}(q_0, x) \in F$.

String Matching Automata

Example

String matching automaton for $p = aba$

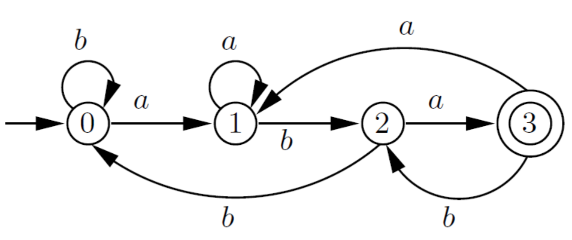


Figure: String matching automaton for $p = aba$

Algorithm for Constructing String Automaton

Definition

Let s, t be strings. If there exist some strings x, y, and z from Σ^* satisfying the conditions

1. $s = xy$,
2. $t = yz$, and
3. $|y|$ is maximal with 1 and 2,

then $\overline{Ov}(s,t) := y$ is called the generalized overlap of s and t. We denote the length of $\overline{Ov}(s,t)$ by $\overline{ov}(s,t)$.

Algorithm for Constructing String Automaton

Definition

Let $p = p_1 \dots p_m \in \Sigma^m$ for an arbitrary alphabet Σ. We define the string matching automaton for p as the finite automaton $M_p = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{0, \dots, m\}$, $q_0 = 0$, $F = \{m\}$, and the transition function δ is defined by

$$\delta(q, a) = \overline{ov}(p_1 \dots p_q a, p) \text{ for all } q \in Q \text{ and } a \in \Sigma.$$

Algorithm for Constructing String Automaton

Algorithm 4.2 Construction of a string matching automaton

Input: A pattern $p = p_1 \ldots p_m$ over an alphabet Σ.

> **for** $q := 0$ **to** m **do**
>   **for all** $a \in \Sigma$ **do**
>   {Compute $\delta(q,a) = \overline{ov}(p_1 \ldots p_q a, p)$}
> $k := \min\{m, q+1\} + 1$
>   **repeat**
> $k := k - 1$
>   **until** $p_1 \ldots p_k = p_{q-k+2} \ldots p_q a$
> $\delta(q,a) := k$

Output: The string matching automaton $M_p = (\{0, \ldots, m\}, \Sigma, 0, \delta, \{m\})$.

Figure: Construction of a string matching automaton

- Running time: $O(\Sigma \cdot m^3)$
- Best running time up to date: $O(\Sigma \cdot m)$

String Matching with Finite Automata

Algorithm 4.3 String matching with finite automata

Input: A text $t = t_1 \dots t_n \in \Sigma^n$ and a pattern $p = p_1 \dots p_m \in \Sigma^m$.

Compute the string matching automaton $M_p = (Q, \Sigma, q_0, \delta, F)$ using Algorithm 4.2.

$q := q_0$

$I := \emptyset$

for $i := 1$ **to** n **do**

  $q := \delta(q, t_i)$

  if $q \in F$ **then**

    $I := I \cup \{i - m + 1\}$

Output: The set I of those positions where p starts as a substring in t.

Figure: String matching with finite automata algorithm

Boyer-Moore Algorithm

Algorithm 4.6 Boyer–Moore algorithm

Input: A pattern $p = p_1 \ldots p_m$ and a text $t = t_1 \ldots t_n$ over an alphabet Σ.

1. Compute from p the function β for the bad character rule.
2. Compute from p the function γ for the good suffix rule.
3. Initialize the set I of positions, where p starts in t, by $I := \emptyset$.
4. Shift the pattern p along the text t from left to right:

$j := 0$
$\gamma(m+1) := m$ {Good suffix rule not applicable for $p_m = t_{j+m}$}
while $j < n - m$ **do**
  {Compare $p_1 \ldots p_m$ and $t_{j+1} \ldots t_{j+m}$ starting from the right}
  $i := m$
  while $p_i = t_{j+i}$ **and** $i > 0$ **do**
    $i := i - 1$
  if $i = 0$ **then**
    $I := I \cup \{j\}$ {The pattern p starts in t at position j}
  {Compute the shift of the pattern according to the bad character rule and the good suffix rule}
  $j := j + \max\{i - \beta(t_{j+i}), m - \gamma(i+1)\}$

Output: The set I of all positions j in t where the pattern p starts.

Figure: Boyer-Moore Algorithm

Boyer-Moore: Best case

What's the best case?

P: **bbbb**
T: **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**
bbbb bbbb bbbb bbbb bbbb bbbb
bbbb bbbb bbbb bbbb bbbb

Every alignment yields immediate mismatch and bad character rule skips *n* alignments

How many character comparisons? *floor(m / n)*

Naive vs Boyer-Moore

As m & n grow, # characters comparisons grows with...

$|P| = n \quad |T| = m$

	Naïve matching	Boyer-Moore
Worst case	m·n	m
Best case	m	m / n

Performance comparison

Simple Python implementations of naïve and Boyer-Moore:

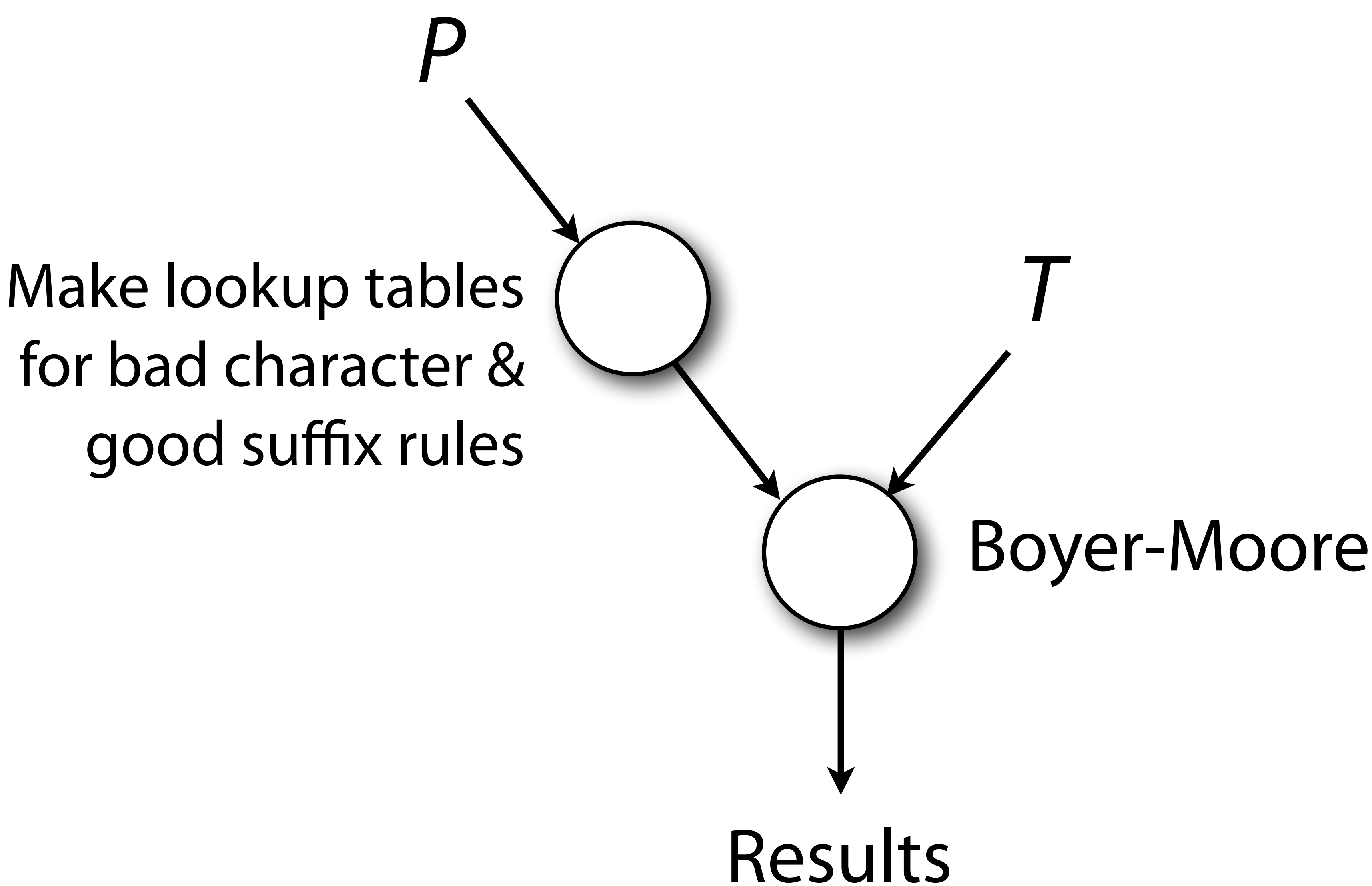
	Naïve matching		Boyer-Moore		
	# character comparisons	wall clock time	# character comparisons	wall clock time	
P: "tomorrow" **T**: Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $\|T\| = 5.59$ M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $\|T\| = 249$ M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

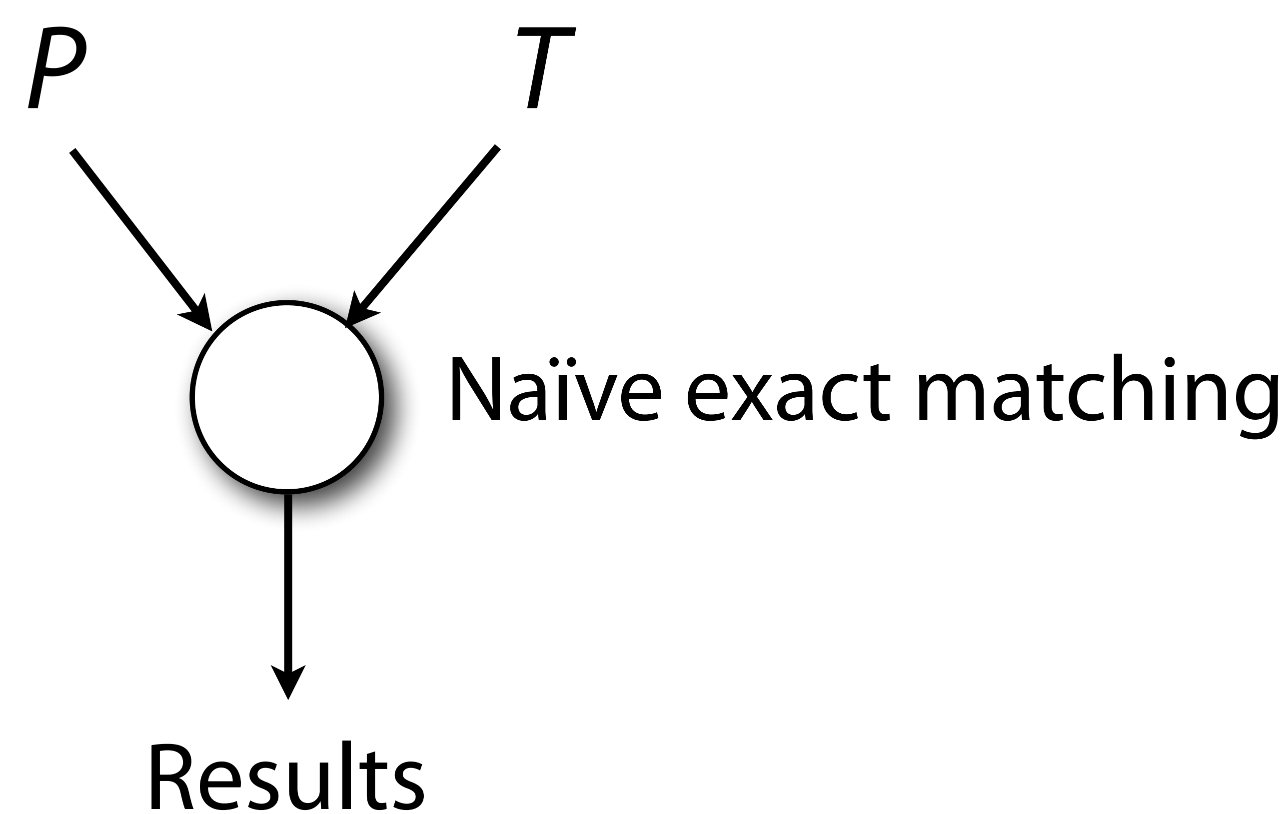
Boyer-Moore implementation

```
def boyer_moore(p, p_bm, t):
    """ Do Boyer-Moore matching """
    i = 0
    occurrences = []
    while i < len(t) - len(p) + 1:  # left to right
        shift = 1
        mismatched = False
        for j in range(len(p)-1, -1, -1):  # right to left
            if p[j] != t[i+j]:
                skip_bc = p_bm.bad_character_rule(j, t[i+j])
                skip_gs = p_bm.good_suffix_rule(j)
                shift = max(shift, skip_bc, skip_gs)
                mismatched = True
                break
        if not mismatched:
            occurrences.append(i)
            skip_gs = p_bm.match_skip()
            shift = max(shift, skip_gs)
        i += shift
    return occurrences
```

Preprocessing: Boyer-Moore



Preprocessing: Naïve algorithm



Preprocessing: Boyer-Moore

Preprocessing: trade one-time cost for reduced work overall via *reuse*

Boyer-Moore preprocesses P into lookup tables that are *reused*

- *reused* for each alignment of P to T_1
- If you later give me T_2, I *reuse* the tables to match P to T_2
- If you later give me T_3, I *reuse* the tables to match P to T_3
- ...

Cost of preprocessing is *amortized* over alignments & texts