Decoding Scheme

- A code word consists of two parts:
  1. **An index**: represents the dictionary entry of the phrase which is the prefix (excluding the last symbol) of the code word.
  2. **A symbol**: the last symbol in the incoming phrase.
- The index of a phrase plays important role in decoder.
- The decoding consists of finding the prefix using index and appending the last symbol to it.

Example of Decoding

- A phrase is stored as tuples $<$index, last_symbol$>$.
- To decode a phrase tuple:
  1. Extract the index from the tuple.
  2. Extract the input phrase using the index.
  3. Obtain the output phrase by appending the tuple symbol to the extracted phrase.
- E.g., say, tuple = (0,A), then index is 0.
- Use index (=0), to retrieve phrase Λ.
- Append Λ to the symbol A and output A as the word.
- For the code word (5, A), extract index=5.
- Retrieve sixth phrase: BB and append A to it.
- So, the output is: BBA.

Decoding Example

(0, A)
(1,A)
(0,B)
(1,B)
(3,B)
(5,A)
(4,C)
(0,C)
(0,D)
(9,C)
(3,A)
(2,A)

Index	Codeword	Output
1	(0, A)	A
2	(1, A)	AA
3	(0, B)	B
4	(1, B)	AB
5	(3, B)	BB
6	(5, A)	BBA
7	(4, C)	ABC
8	(0, C)	C
9	(0, D)	D
10	(9, C)	DC
11	(3, A)	BA
12	(2, A)	AAA

Decompressed Message:
AAABABBBBBAABCDDCBAAAA

Decoder Algorithm

```
while (true) {
    wait for the next code word <i, s>;
    decode phrase = dictionary[i].s;
    add phrase to dictinary;
}
```

Analysis of Compression

- Let us work out the worst case for the binary alphabets.
- We ask the question:
  - What is the maximum number of distinct phrases in a string of length at most k?
  - For $k = 1$ it would 2: 0|1.
  - For $k = 2$ it would 6: 0|1|00|01|10|11.
  - For $k = 3$ it would 2*1+4*2+8*3 = 34.
- In general, for a length k, string length is:

$$n_k = \sum_{j=1}^{k} j * 2^j$$

- Use induction to prove that $n_k = (k-1)2^{k+1} + 2$.

Analysis of Compression

- Let $c(n_k)$: number of distinct phrases for string of length n_k.

$$\begin{aligned} c(n_k) &= \sum_{i}^{k} 2^i = 2^{k+1} - 2 \\ &\leq \frac{(k-1)2^{k+1}}{k-1} \\ &\leq \frac{n_k}{k-1} \end{aligned}$$

- For an arbitrary length n, assume $n = n_k + \Delta$.
- Also, since $c(n_k) < 2^{k+1}$, $\log c(n_k) < k + 1$.

Analysis of Compression

- When $c(n)$ becomes maximum?
- The first n_k bits parsed into $c(n_k)$ phrases.
- The remaining Δ bits can be parsed into phrases of length $k+1$.
- So, for an arbitrary length n of the string, the number of distinct phrases can be at most:

$$
\begin{aligned}
c(n) &\leq \frac{n_k}{k-1} + \frac{\Delta}{k+1} \\
&\leq \frac{n_k + \Delta}{k-1} \\
&\leq \frac{n}{k-1}, \text{ where } n_k \leq n < n_{k+1} \\
&\leq \frac{n}{\log c(n) - 3}, \text{ as } c(n) = 2^{k+1} - 2
\end{aligned}
$$

Analysis of Compression

- In general, when a bit string broken into $c(n)$ phrases:
  - Each phrase requires $\log c(n)$ index bits
  - Each symbol requires $\log \alpha$ bits, where $\alpha = |\Sigma| = 2$ (in case of binary alphabets).
  - Overall size of compression: $c(n)(\log c(n) + 1)$ bits.

  From the previous inequality, we derive:

$$
\begin{aligned}
c(n)\log c(n) + c(n) &\leq c(n)(\log c(n) - 3) + 3c(n) + c(n) \\
&\leq \frac{n}{\log c(n) - 3}(\log c(n) - 3) + 4c(n) \\
&\leq n + 4c(n) \\
&= n + O\left(\frac{n}{\log n}\right) = O(n)
\end{aligned}
$$

- This implies that we don't require more than n bits to compress any string of length n

String Matching

- Finding pattern in strings is important in context of text editing.
- The problem is defined as follows:
  - Given a pattern P and a text T find all occurrences of P in T.
  - The alphabet set from which P and T are constructed is assumed to be finite.
- As output we are interested in positions in T from where the string P occurs:
  - It specifies values of shifts s, where $0 \leq s \leq n - m$ such that $P[1, \ldots, m] = T[s + 1, \ldots, s + m]$
  - In other words, $P[j] = T[s + j]$, for $1 \leq j \leq m$ and $0 \leq s \leq n - m$.

A Simple Minded Approach

- Start matching from $T[s+1]$, where $s = 0$, in each corresponding position.
- If a mis-match occurs slide P one position to the right and restart matching,i.e., increment s by 1.
- This algorithm obviously works, but very expensive.
- The maximum value of s will be $n - m$. So, s changes $n - m + 1$ times.
- After each shift at most O(m) character comparisions may be needed.
- So the running time is O($m.(n - m + 1)$).

Boyer Moore Algorithm

- Boyer Moore speeds up the matching by sliding P in large steps.
- Character-wise comparison is performed from right to left.
- It uses clever shifting rules for matching P against the target text T.

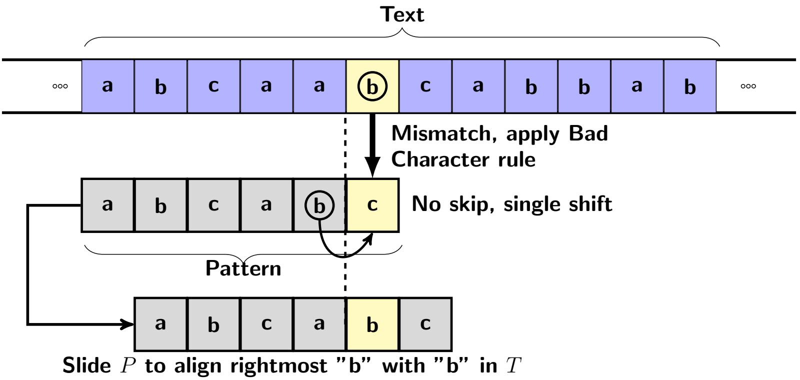
Top Level of Description Boyer Moore Method

1. Place P initially to align $P[1 \ldots m]$ with $T[1 \ldots m]$.
2. Scan from the right starting comparison of $T[m] : P[m]$.
3. If a mismatch occurs at j, say $T[j] = x$ and $P[j] = y$, and $x \neq y$, then using two rules to place P appropriately aligned under T.
4. Repeat from Step 2.

Cleverness of Boyer Moore matching lies in step 3. It uses two rules.

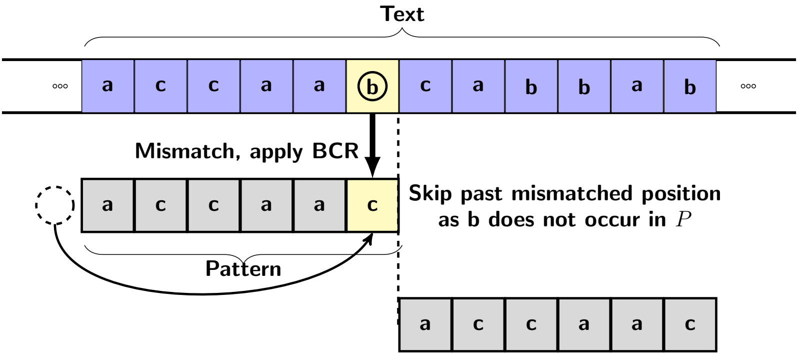
- Bad Character Rule & Good Suffix Rule

Rule 1: Bad Character Rule

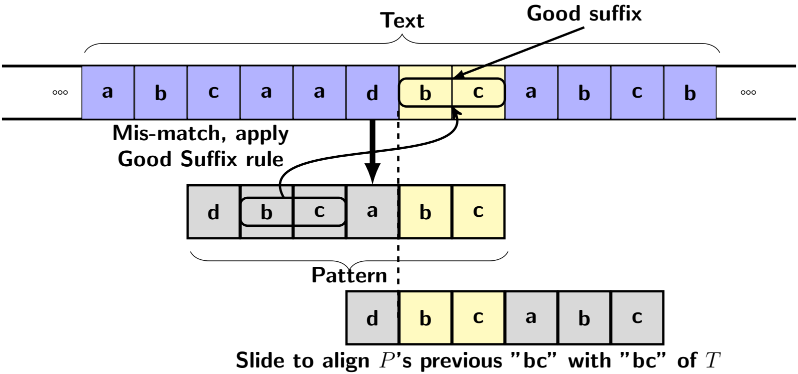


Bad Character Rule

- Let the mismatched position be k in P, and the corresponding characters be $P[k] = x$ and $T[s+k-1] = y$.
- Let the rightmost y in P occur at $P[j]$, where $1 \leq j \leq k-1$.
- Then align $P[j]$ (=y) with $T[s+k-1]$ (=y), and restart match from thereon.
- If no j, $1 \leq j \leq k-1$ is found such that $P[j] = y$, then shift over to align $P[1]$ with $T[s+m+1]$.


Text
ooo
a
c
c
a
a
b
c
a
b
b
a
b
ooo
Mismatch, apply BCR
a
c
c
a
a
c
Skip past mismatched position
as b does not occur in P
Pattern
a
c
c
a
a
c

Good Suffix Rule



Good Suffix Rule

- This rule is a generalization of BCR but slightly complicated.
- Let t be a substring of T matched with a suffix of P then skip until any one of the following is met first.
  1. t matches with corresponding characters of P, or
  2. A prefix of P matches with a suffix of t, or
  3. P moves past t.

Good Suffix Rule

t

T: C G T G C C 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 matching a few past symbols in P

t

T: C G T G C C 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 with suffix of t

T: C G T G C C 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

Execution of BM Algorithm

- The shifting of P over T is done with use of both rules.
- The rules can be used independently.
- Boyer Moore uses both the rules by taking the maximum of the shifts obtained from each.
- It then applies the maximum shift in aligning P with T.
- In worst case, only single shift would be possible.
- Therefore, in theory its complexity is same as simple string matching algorithm.
- In practice, it is found to be very fast.

Execution of BM Algorithm

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   gsr:0, bcr:6

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   gsr:2 [case(a)], bcr:0

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   gsr:7 [case(b)], bcr: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