Exact Matching Algorithms

Michael T. Goodrich
University of California, Irvine

Some slides adapted from https://www.cs.bgu.ac.il/~dinitz/Course/SS-12/Boyer-Moore-algorithm-Vladimir.pptx

Review: Strings

- A **string** is a sequence of characters (indexed from 0)*
- Examples of strings:
  - Python program
  - HTML document
  - DNA sequence
  - Digitized image
- An **alphabet** Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII or Unicode
  - {0, 1}
  - {A, C, G, T}

***Some people index starting from 1.**

- Let P be a string of size m
  - A **substring** $P[i:j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
  - A **prefix** of P is a substring of the type $P[0:i]$
  - A **suffix** of P is a substring of the type $P[i:m-1]$
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P
- Applications:
  - Text editors
  - Search engines
  - Biological research

Application: fgrep

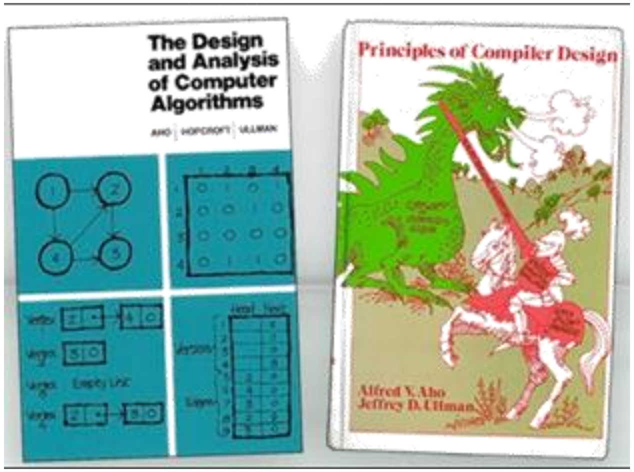
- Recall that **fgrep** looks for an exact match of a text string in a file.
- So we are interested in fast algorithms for the **exact match** problem:
    - Given a text string, T, of length n, and a pattern string, P, of length m, over an alphabet of size k, find the first (or all) places where a substring of T matches P.

```
    01234567890123456789012345678
S = HACKHACKHACKHACKITHACKEREARTH
P =     HACKHACKIT
P =         HACKHACKIT...[match!]
P =                   HACKHACKIT
```

Image from https://www.hackerearth.com/practice/notes/exact-string-matching-algorithms/

Alfred Aho

- 1975: Invented **fgrep**
- …*
- 2020: received the Turing Award



* Also invented text processing techniques used in every modern source-code compiler and co-authored two influential textbooks.

Images from https://awards.acm.org/about/2020-turing

Brute-force Pattern Matching

- The Brute-force (Naïve) pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences

Algorithm ***BruteForceMatch***(T, P)

  Input text T of size n and pattern P of size m

  Output starting index of a substring of T equal to P or -1 if no such substring exists

  for $i \leftarrow 0$ **to** $n - m$

    { test shift i of the pattern }

    $j \leftarrow 0$

    while $j < m \wedge T[i + j] = P[j]$

      $j \leftarrow j + 1$

    if $j = m$

      return i {match at i}

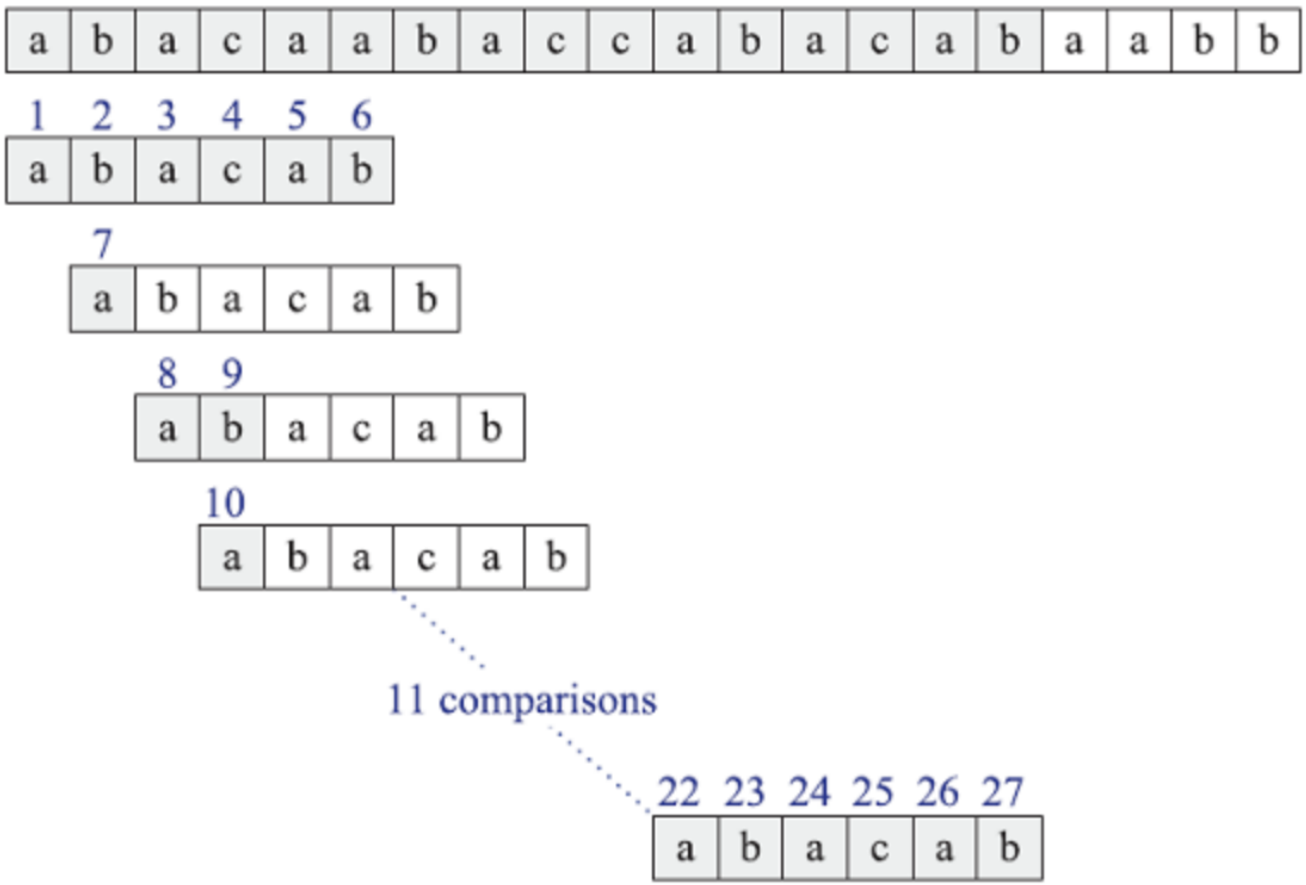
    else

      break while loop {mismatch}

  return -1 {no match anywhere}

Brute-Force Matching Example

- Trying every possible position for a match:



Expected-case Analysis for Brute-force

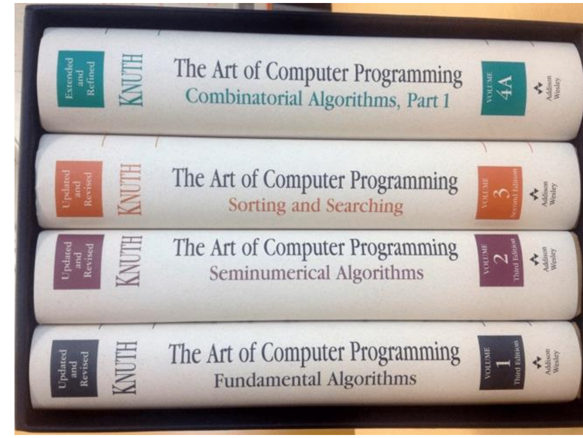
- The worst-case running time for Brute-force algorithm O(mn), but it runs in expected linear time for random strings.
- Suppose P and T are strings of m and n characters respectively chosen uniformly and independently at random from an alphabet of size k.
- Let $X_{i,j}$ be a random variable that is 1 if and only if P[i] is compared to T[j], and note that probability $X_{i,j}$ is 1 is $1/k^i$ because this occurs when we have i character matches.
- By the linearity of expectation, the expected number of comparisons for any T[j] is therefore

$$1/k + 1/k^2 + 1/k^3 + \ldots + 1/k^m,$$

which is at most 2.

- Thus, the expected number of comparisons is at most 2n.

Donald Knuth

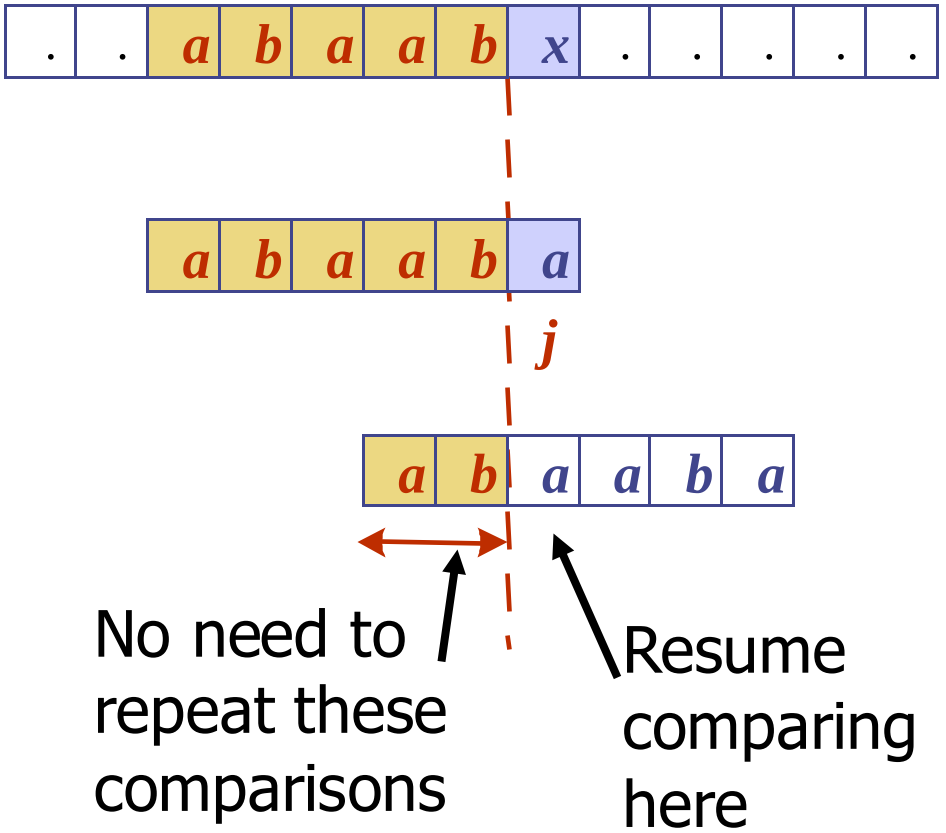


- 1973: Discovered the KMP algorithm (which was also published in a technical report by Morris and Pratt in 1970—all three published a joint paper describing the algorithm in 1977).
- 1974: Received the Turing Award.
- He is also known for his book series, “The Art of Computer Programming,” which formalized and popularized algorithm analysis (e.g., the “big O”).

Image from https://en.wikipedia.org/wiki/Donald_Knuth

The KMP Algorithm

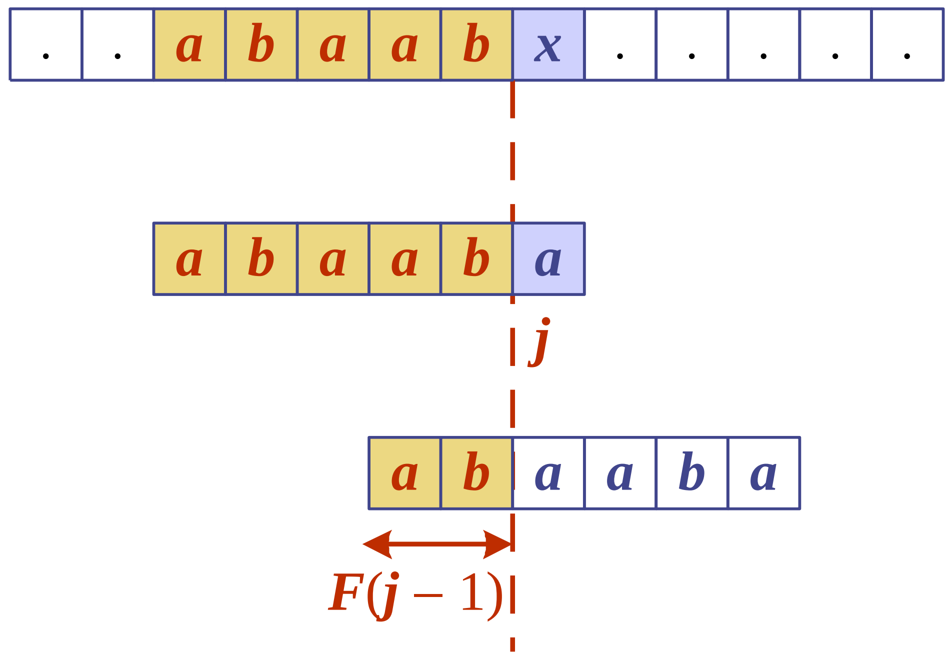
- Consider the comparison of a pattern with a text as in the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the **largest** prefix of $P[0..j]$ that is a suffix of $P[1..j]$
- This approach is similar to the NFA-to-DFA approach, but is implemented more efficiently.



The KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the length of the longest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ and $j > 0$, we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3



The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```

Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

Algorithm ***failureFunction***(***P***)

```
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

Example

a	*b*	*a*	*c*	*a*	*a*	*b*	*a*	*c*	*c*	*a*	*b*	*a*	*c*	*a*	*b*	*a*	*a*	*b*	*b*

1	2	3	4	5	6
a	*b*	*a*	*c*	*a*	*b*

	7				
a	*b*	*a*	*c*	*a*	*b*

8	9	10	11	12	
a	*b*	*a*	*c*	*a*	*b*

13					
a	*b*	*a*	*c*	*a*	*b*

14	15	16	17	18	19
a	*b*	*a*	*c*	*a*	*b*

j	0	1	2	3	4	5
***P*[*j*]**	***a***	***b***	***a***	***c***	***a***	***b***
***F*(*j*)**	0	0	1	0	1	2

The Boyer-Moore-Horspool Algorithm

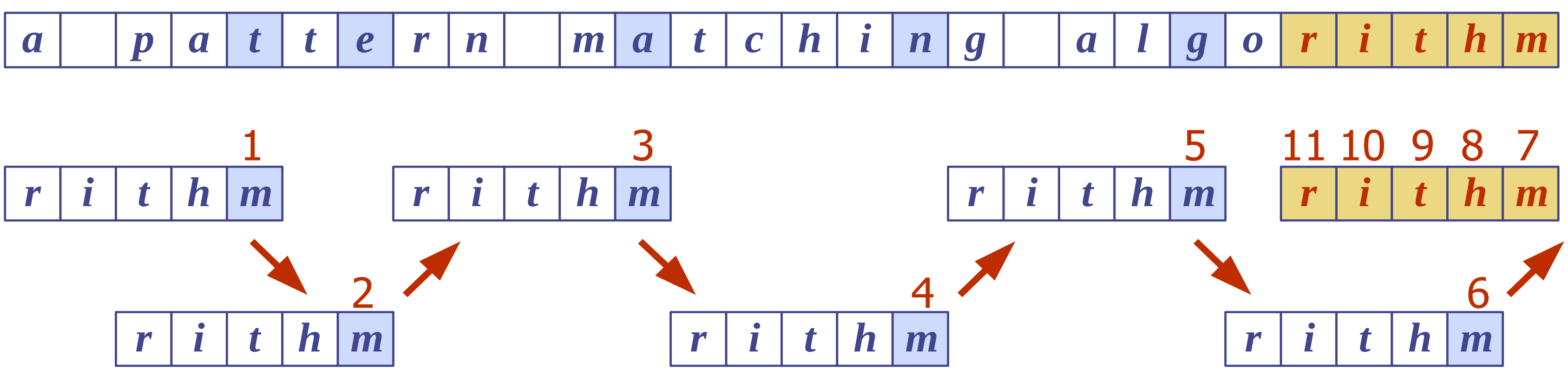
- The Boyer-Moore-Horspool algorithm for pattern matching a pattern P of length m in a text of length n is based on the following two simple heuristics:

  Reverse-match heuristic: Compare $\boldsymbol{P}$ with a subsequence of $\boldsymbol{T}$ moving backwards

  Bad-character heuristic: When a mismatch occurs at $\boldsymbol{T}[\boldsymbol{i}] = \boldsymbol{c}$

  - If $\boldsymbol{P}$ contains $\boldsymbol{c}$, shift $\boldsymbol{P}$ to align the last occurrence of $\boldsymbol{c}$ in $\boldsymbol{P}$ with $\boldsymbol{T}[\boldsymbol{i}]$
  - Else, shift $\boldsymbol{P}$ to align $\boldsymbol{P}[0]$ with $\boldsymbol{T}[\boldsymbol{i}+1]$

- Example:



Last-Occurrence Function

- The Boyer-Moore-Horspool algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

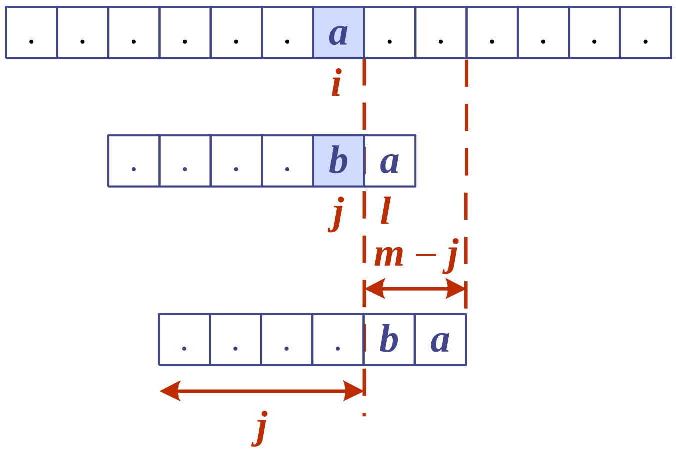
c	a	b	c	d
$L(c)$	4	5	3	-1

- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + k)$, where m is the size of P and k is the size of Σ. **How?**

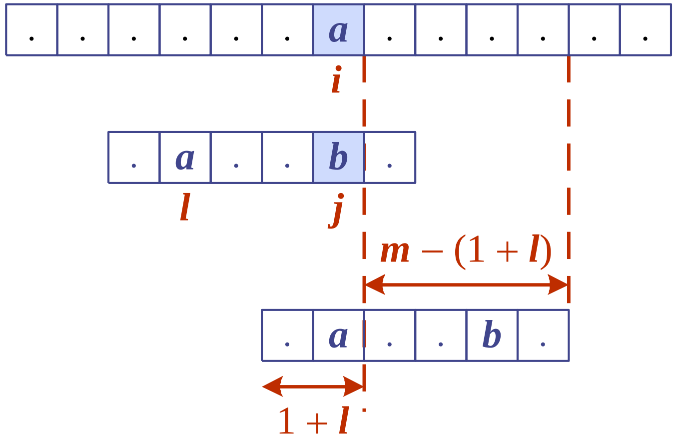
The Boyer-Moore-Horspool Algorithm

Algorithm *BoyerMooreHorspool*(T, P, Σ)
$L \leftarrow$ *lastOccurenceFunction*(P, Σ)
$i \leftarrow m - 1$
$j \leftarrow m - 1$
repeat
  if $T[i] = P[j]$
    if $j = 0$
      return i { match at i }
    else
      $i \leftarrow i - 1$
      $j \leftarrow j - 1$
  else
    { bad-character-jump }
    $l \leftarrow L[T[i]]$
    $i \leftarrow i + m - \min(j, 1 + l)$
    $j \leftarrow m - 1$
until $i > n - 1$
return -1 { no match }

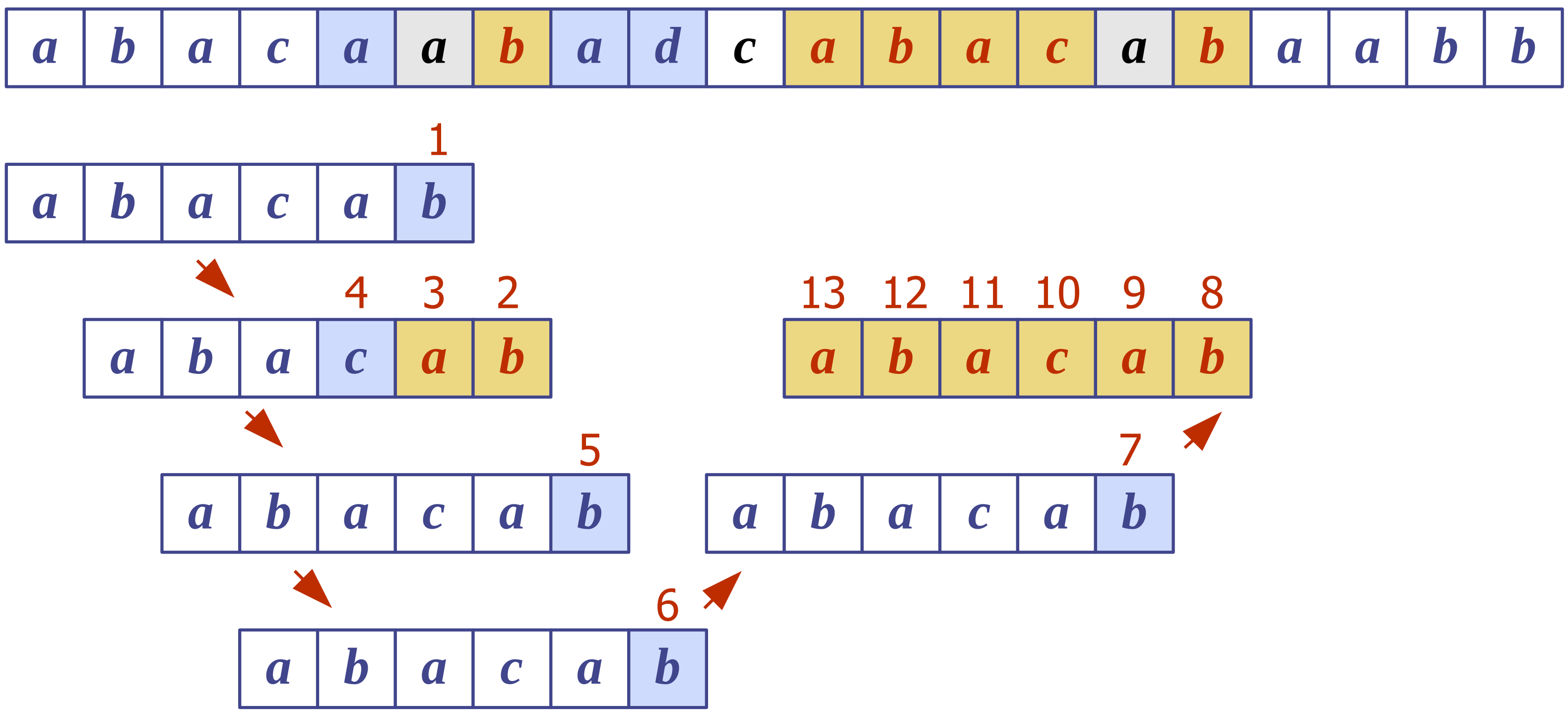
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$



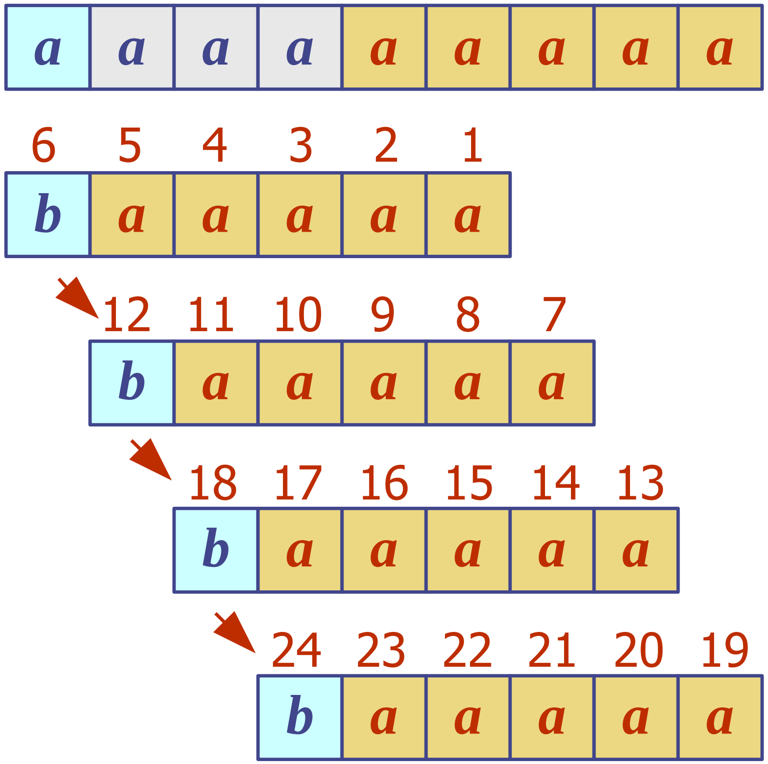
Example



a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

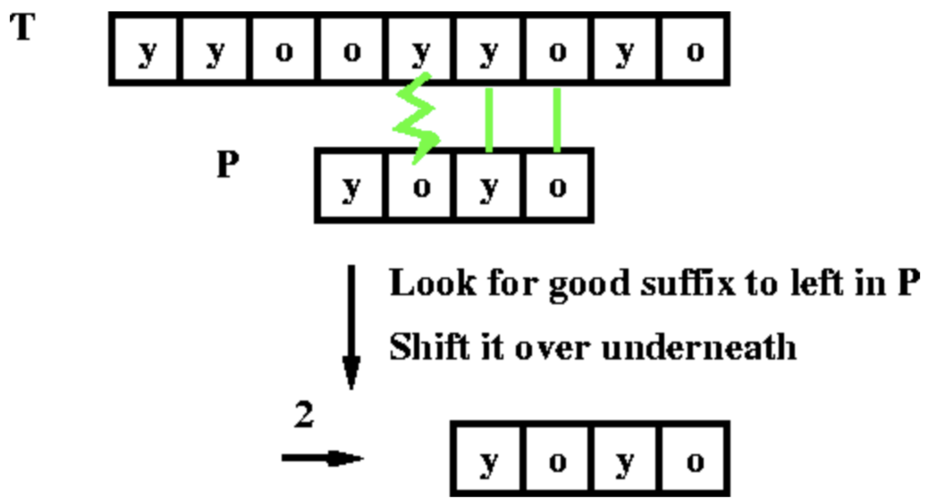
Analysis

- The Boyer-Moore-Horspool algorithm runs in $O(nm + k)$ time in the **worst case**
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- The Boyer-Moore-Horspool algorithm can skip over some comparisons
- It runs in O(n/m + m) time in the best case.



The Boyer-Moore Algorithm

- The original Boyer-Moore has another heuristic, the **good suffix rule:**



- When a mismatch occurs, we take the biggest shift possible using the bad character and good suffix rules

Image from https://eecs.wsu.edu/~cook/aa/lectures/l24/node16.html

Case 1 for the good suffix rule

- Suppose we have already matched a suffix, u, of P, and u appears in P. Then we want a shift that is guaranteed to match u and requires the mismatching character to be different:

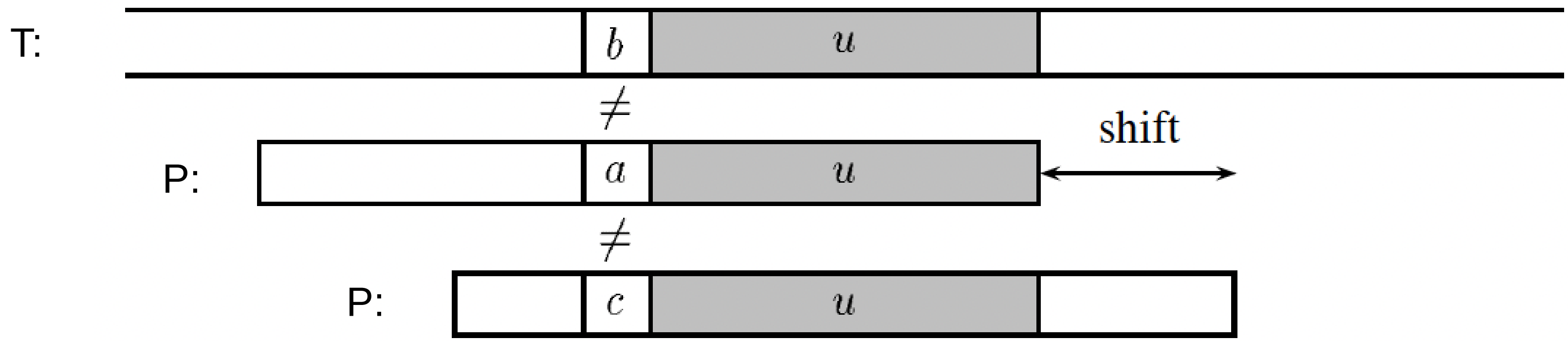


Image from https://doi.org/10.1016/S1570-8667(03)00005-4

Case 2 for the good suffix rule

- Suppose we have already matched a suffix, u, of P, and u does not appear in P. Then we want a shift such that a prefix v is a suffix of u:

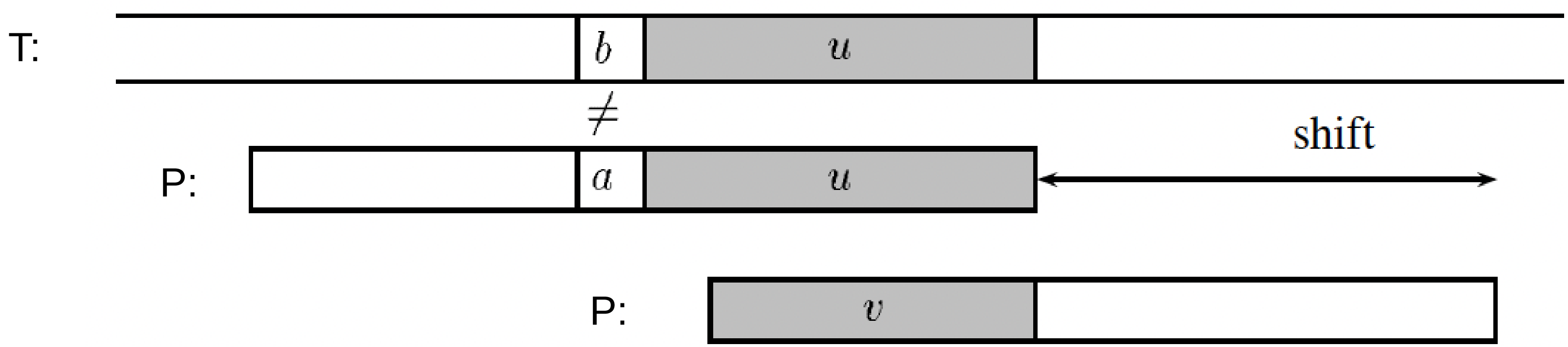


Image from https://doi.org/10.1016/S1570-8667(03)00005-4

Good Suffix Rule

- *Definition:* Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs to the next character to the left. Then find, if exists, the **rightmost** copy t' of t in P, such as t' is not a suffix of P and the character to the left of t' in P differs from the character to the left of t in P. Shift P to the right, so that substring t' in P is below substring t in T.

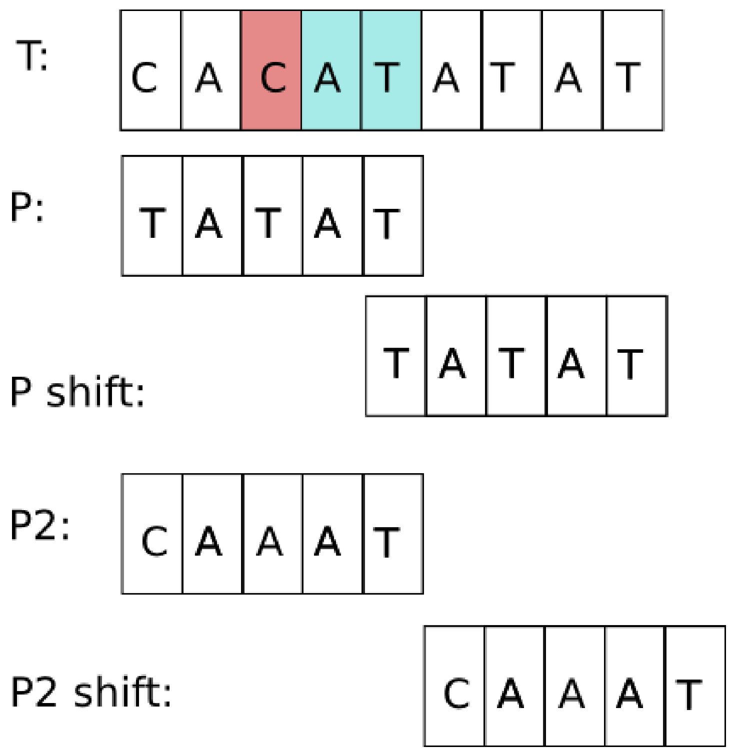
M	A	N	P	A	N	A	M	A	N	A	P	-
A	N	A	M	P	N	A	M	-	-	-	-	-
-	-	-	-	A	N	A	M	P	N	A	M	-

Demonstration of good suffix rule with pattern **ANAMPNAM**.

Image from https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm

Good suffix rule (cont'd)

- If t' does not exist, then shift the left end of P past the left end of t in T by the **least** amount, so that a prefix of P matches a suffix of t in T. If no such shift is possible then shift P by m places to the right.



The good suffix shift table

- Define a shift table, MATCH(i), that encodes the good suffix shifts for a pattern, x, of length m.
  - MATCH(i) = min. s such that Cs(i,s) and Cos(i,s) hold:

$$Cs(i,s) = \begin{cases} 0 < s \leq i \text{ and } x[i-s+1\,..\,m-s-1] \text{ is a suffix of } x \\ \text{or} \\ s > i \text{ and } x[0\,..\,m-s-1] \text{ is a suffix of } x \end{cases}$$

$$Cos(i,s) = \begin{cases} 0 \leq s \leq i \text{ and } x[i-s] \neq x[i] \\ \text{or} \\ s > i \end{cases}$$

Images from https://doi.org/10.1016/S1570-8667(03)00005-4

The bad character shift table

- Let x be a pattern of length m.
- Define a bad-character shift table, occ[a], for each character, a, in the alphabet for x:

$$occ[a] = \begin{cases} \min\{i \mid 1 \le i \le m-1 \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

- This is just the last occurrence function, L, indexed slightly differently.
  - It can be computed in the same way as L.

Image from https://doi.org/10.1016/S1570-8667(03)00005-4

The Boyer-Moore Algorithm

- Let x be a pattern of length m and y a text of length n.
  - j is the location of a possible match.

```
BOYER-MOORE(x, m, y, n)
j ← 0
while j ≤ n − m
      do i ← m − 1
         while i ≥ 0 and x[i] = y[i + j]
               do i ← i − 1
         if i < 0
            then REPORT(j)
                 j ← j + MATCH(0 )
            else j ← j + max(MATCH(i ), occ[y[i + j]] − m + i + 1)
```

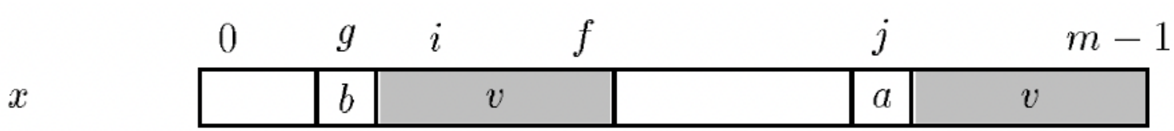
Algorithm description from https://doi.org/10.1016/S1570-8667(03)00005-4

Computing the Suffix table

- Compute a table, suf, such that suf[i] is the length of the longest suffix of x ending at position i in x.
  - We can compute the suf table like the KMP Failure function, but in reverse (j = g+m-1-f):

SUFFIXES(x, m)

```
suf[m − 1] ← m
g ← m − 1
for i ← m − 2 downto 0
    do if i > g and suf[i + m − 1 − f] < i − g
          then suf[i] ← suf[i + m − 1 − f]
          else g ← min{g, i}
               f ← i
               while g ≥ 0 and x[g] = x[g + m − 1 − f]
                   do g ← g − 1
               suf[i] ← f − g
return suf
```



Algorithm description from https://doi.org/10.1016/S1570-8667(03)00005-4

Computing the MATCH table

- Given the suffix table, suf, we compute MATCH to be sMatch in the following algorithm:

```
STRONG-MATCHING(x, m)
j ← 0
for i ← m − 1 downto −1
    do if i = −1 or suf[i] = i + 1
        then while j < m − 1 − i
                 do sMatch[j] ← m − 1 − i
                    j ← j + 1
for i ← 0 to m − 2
    do sMatch[m − 1 − suf[i]] ← m − 1 − i
return sMatch
```

Algorithm description from https://doi.org/10.1016/S1570-8667(03)00005-4

Summary for the Boyer-Moore Algorithm

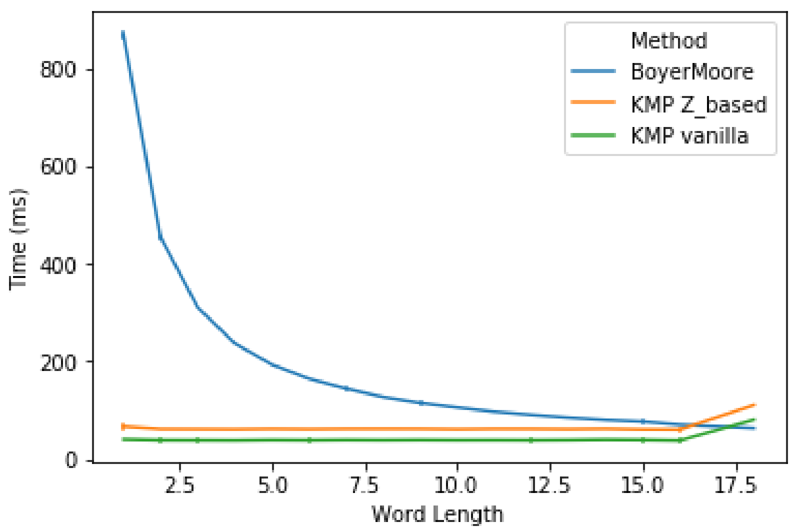
- The Boyer-Moore algorithm runs in $O(n + m)$ time in the worst case.
- It runs in $O(n/m + m)$ time in the best case.
- It can be further optimized to find all occurrences of a pattern in a text using at most $1.5n$ character comparisons.

Experimental Analysis

- Since completely random strings are not useful for analyzing exact string-matching algorithms, we need alternatives:
  - **Seeded random strings**: Create a random text string, T, of length n (e.g., n=1,000,000), and a random pattern, P, of length m (e.g., m=5, 10, 20, …). Then insert P into T at 1 to100 random locations.
  - **English text**: Use a corpus of large English text (e.g., emails) and search for patterns of various lengths (e.g., email addresses, English words, English phrases).

Varying the Pattern Length

- One type of experiment: Keep the text size fixed at a reasonably large amount and vary the pattern size.



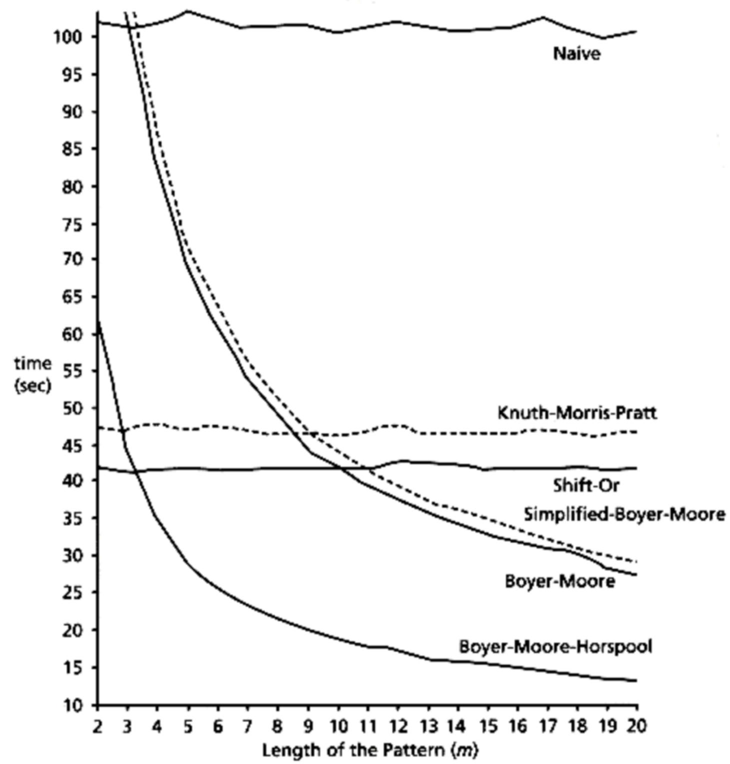


Figure 10.12: Simulation results for all the algorithms in English text

Images from https://dearxxj.github.io/post/4/, http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap10.htm

Varying the Text Length

- Another type of experiment: Vary the text length, n, with certain pattern lengths (e.g., m=10, 20, 100) or as a function of n (e.g., $m=n^{1/2}$ or m=n/8).

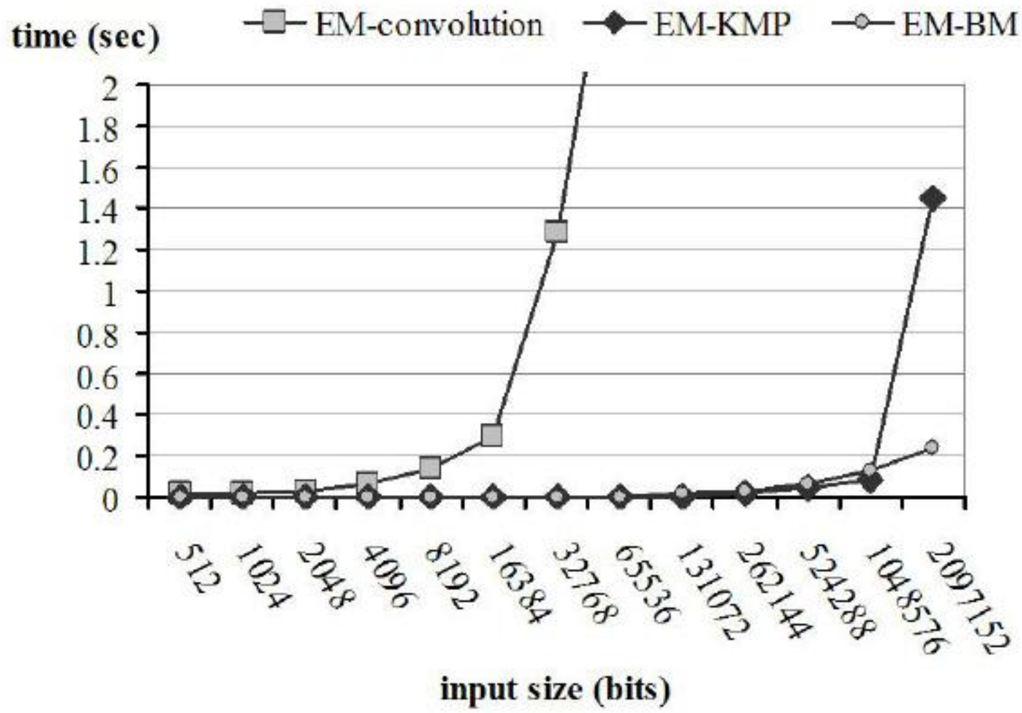


Image from https://www.researchgate.net/figure/A-comparison-of-the-convolution-KMP-and-Boyer-Moore-algorithms-for-the-exact-matching_fig2_220444712

Data Type Duality

- Rather than rely only on comparing characters, numerical matching algorithms take advantage of the fact that characters in a string can also be viewed as (binary) numbers.
- This concept is referred to as **data type duality**.

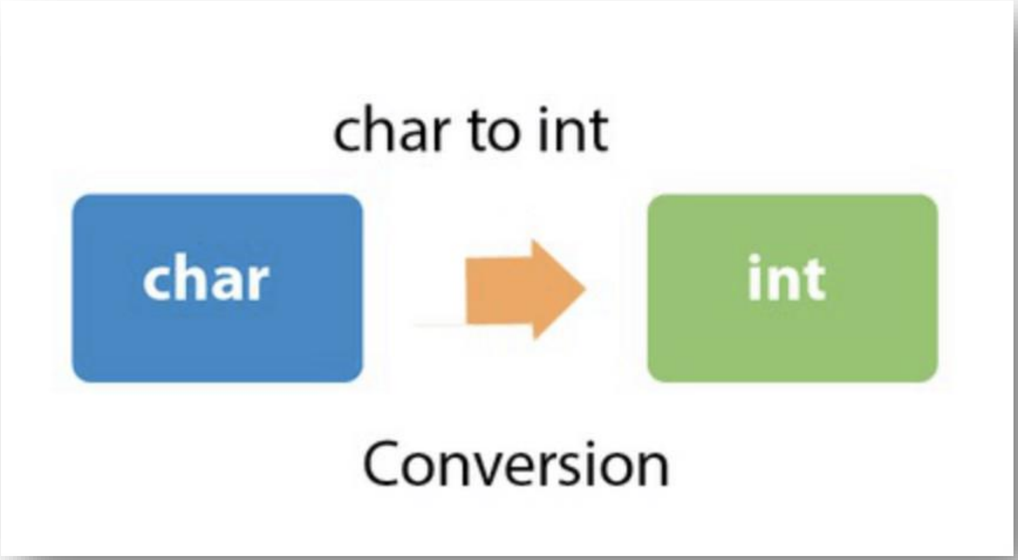
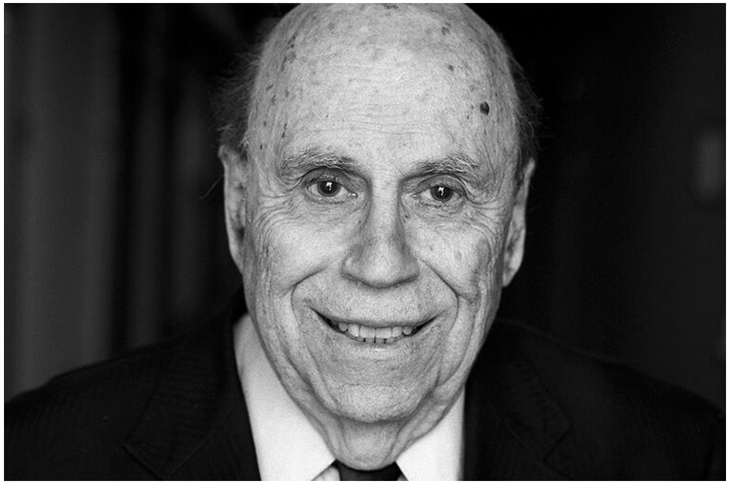


Image from https://www.javatpoint.com/java-char-to-int

The Rabin-Karp Algorithm

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character substring of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence at this location (in case of a hash value collision causing a false match).
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.
- (Recall that we highlighted Michael Rabin in a previous lecture.)

Michael Rabin



- 1959: Invented nondeterministic finite automata and introduced polynomial time as a notion of algorithm efficiency
- 1976: Received the Turing Award.
- 1987: Developed the Rabin-Karp string searching algorithm with Richard Karp.

Image from https://www.heidelberg-laureate-forum.org/laureate/michael-o-rabin/

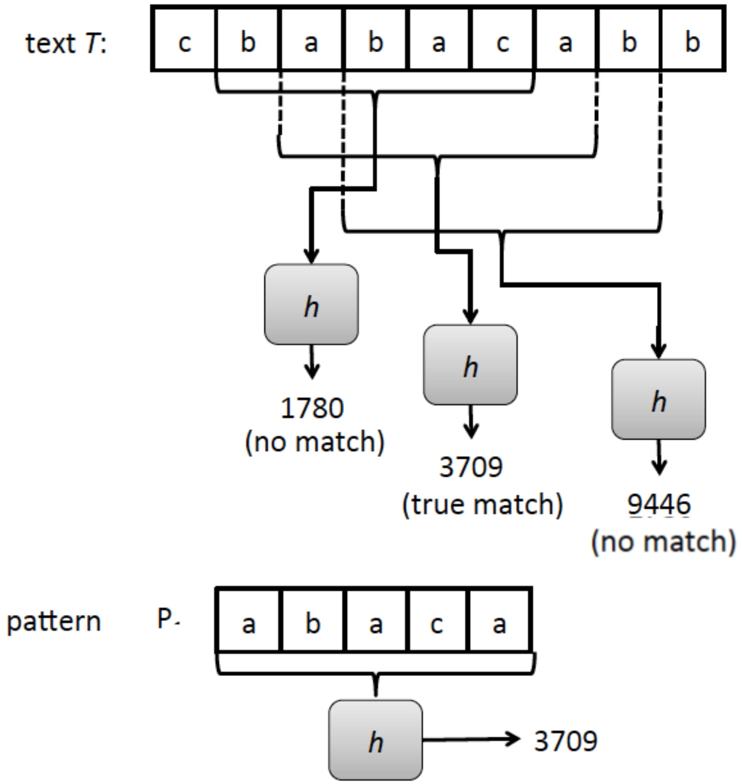
Richard Karp

- 1985: Received the Turing Award.
- 1987: Developed the Rabin-Karp string searching algorithm with Michael Rabin.
- He is also known for publishing a landmark paper proving 21 problems to be NP-complete.
- The PhD advisor of UCI Professor Sandy Irani.

Image from https://en.wikipedia.org/wiki/Richard_M._Karp

Rabin-Karp Example

- Text T = cbabacabb
- Pattern P = abaca



Rabin-Karp Algorithm (High Level)

text is n characters long, pattern is m characters long

hash_p=hash value of pattern

hash_t=hash value of first m letters in text

repeat

    if (hash_p == hash_t)

        do brute force comparison of pattern and selected section of text

    hash_t = hash value of next section of text, one character over

until (end of text **or** brute force comparison == true)

- Running time is O(nm) if we recompute hash_t for each substring of m characters in the text, which is no better than brute-force matching!

Rabin-Karp Rolling Hash Function

- We can do better by using a rolling hash function, which allows us to compute each hash value from the previous hash value.
- Consider an m-character sequence as an m-digit number in base b, where b is the number of letters in the alphabet. The text subsequence t[i : i+m-1] is mapped to the number

$$x(i) = t[i]\cdot b^{M-1} + t[i+1]\cdot b^{M-2} + \ldots + t[i+M-1]$$

Given x(i) we can compute x(i+1) for the next substring t[i+1 : i+M] in constant time:

$$x(i+1) = t[i+1]\cdot b^{M-1} + t[i+2]\cdot b^{M-2} + \ldots + t[i+M]$$

$x(i+1) = x(i)\cdot b$ Shift left one digit

$- t[i]\cdot b^{M}$ Subtract leftmost digit

$+ t[i+M]$ Add new rightmost digit

Polynomial Rolling Hash Function

- The original Rabin-Karp algorithm used the a standard polynomial hash function:

$$H = c_1 a^{k-1} + c_2 a^{k-2} + c_3 a^{k-3} + \ldots + c_k a^0,$$

where a is a constant, and $c_1, \ldots, c_k$ are the input characters

  - This requires 2 multiplications and an addition and subtraction to compute each new hash value.
- Multiplications are generally slower than comparing characters, and these multiplications are in the “inner loop” of the algorithm.
- So it may be helpful to have a different hash function.

Bitwise Operators

- Typical built-in bitwise (bit-parallel) operators, which are faster than multiplication:

<table>
<tr><th>Operator</th><th>Example</th><th>Meaning</th></tr>
<tr><td>&</td><td>a & b</td><td>Bitwise AND</td></tr>
<tr><td>|</td><td>a | b</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>a ^ b</td><td>Bitwise XOR (exclusive OR)</td></tr>
<tr><td>~</td><td>~a</td><td>Bitwise NOT</td></tr>
<tr><td><<</td><td>a << n</td><td>Bitwise left shift</td></tr>
<tr><td>>></td><td>a >> n</td><td>Bitwise right shift</td></tr>
</table>

Image from https://realpython.com/python-bitwise-operators/

Examples

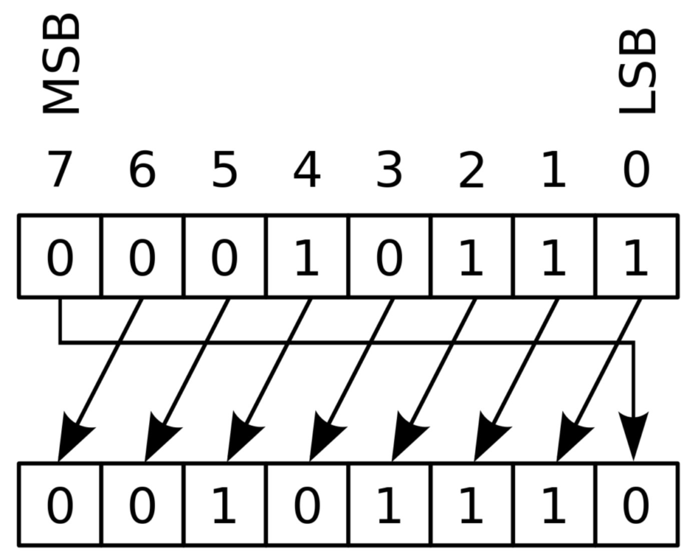
- Bitwise operations:

Number 1	1	0	1	0	1
Number 2	1	1	1	0	0
AND	1	0	1	0	0
OR	1	1	1	0	1
XOR	0	1	0	0	1

Note the following:

- X AND X =X
- X OR X = X
- X XOR X = 0

Cyclic shift by 1 bit:

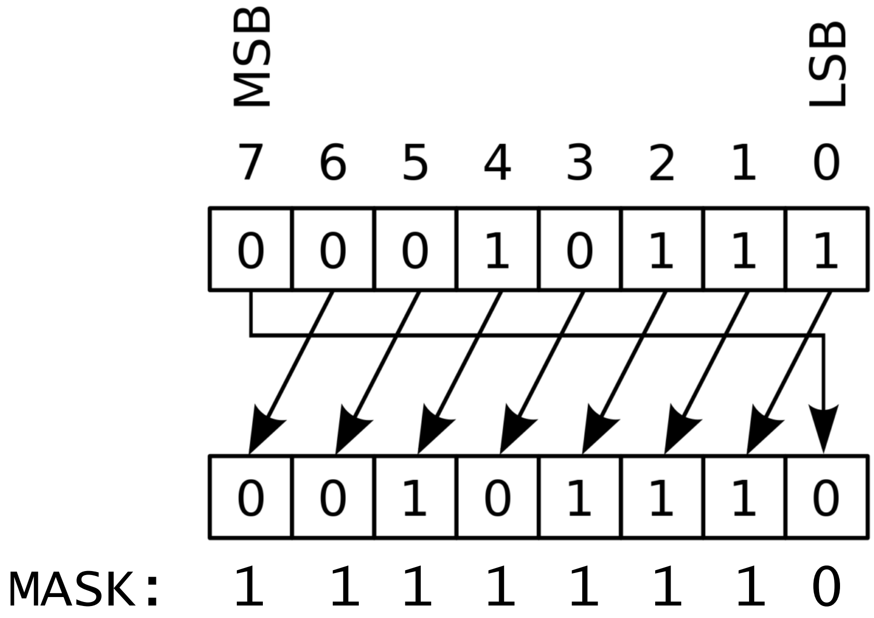


Note that bit vectors are indexed from right to left.

Typical Syntax for Cyclic Shift

- To do a cyclic shift by k bits in C (assumes k < Integer.SIZE):
  - return (bits << k & MASK) | (bits >> (Integer.SIZE - k))

Cyclic shift by 1 bit:



Cyclic Polynomial Hash Function

- Let the function s be a cyclic binary rotation (or circular shift): it rotates the bits by 1 to the left, pushing the leftmost bit around to the first position.
  - E.g., s(101)=011, s(101)=011.
- Define the hash H as follows, where ⊕ is XOR and h is a random hash function (or lookup table):

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k)$$

- The new hash value (2 shifts and 2 XORs):

$$H \leftarrow s(H) \oplus s^k(h(c_1)) \oplus h(c_{k+1}),$$

where c_{k+1} is the new character.

The Rabin-Karp Algorithm

- Assumes a shiftHash(f, T, i) function for computing a shifted rolling hash value for position i in T given the hash value, f, for position i-1 in T.

Let H be the hash of the pattern, i.e., H = h(P)

for i ← 0 to n − m do

    if i = 0 **then** // initial hash

        f ← h(T[0 : m − 1])

    else

        f ← shiftHash(f, T, i)

        if f == H **then**

            // check P against T[i : i + m − 1]

            j ← 0

            while j < m **and** T[i + j] = Pk[j] **do**

                j ← j + 1

                if j = m **then**

                    return j as a match location

Analysis of the Rabin-Karp Algorithm

- We are given a test of length n and a pattern of length m.
- Use a hash function that is random enough so the probability of a false match is at most 1/m.
- Then the expected running time to find a first match for the pattern (if it exists) is O(n+m).