Pattern 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

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}
- 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

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

Application: fgrep

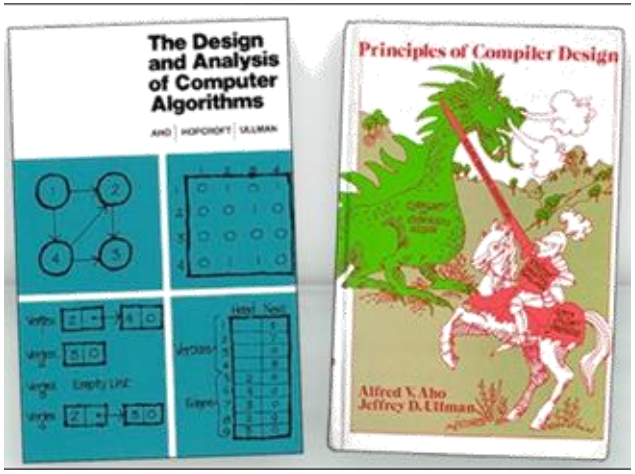
- Recall that **fgrep** looks for an exact match of a text string in a file.
- So we are interested in fast algorithms for the **exact match** problem:
  - Given a text string, T, of length n, and a pattern string, P, of length m, over an alphabet of size k, find the first (or all) places where a substring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

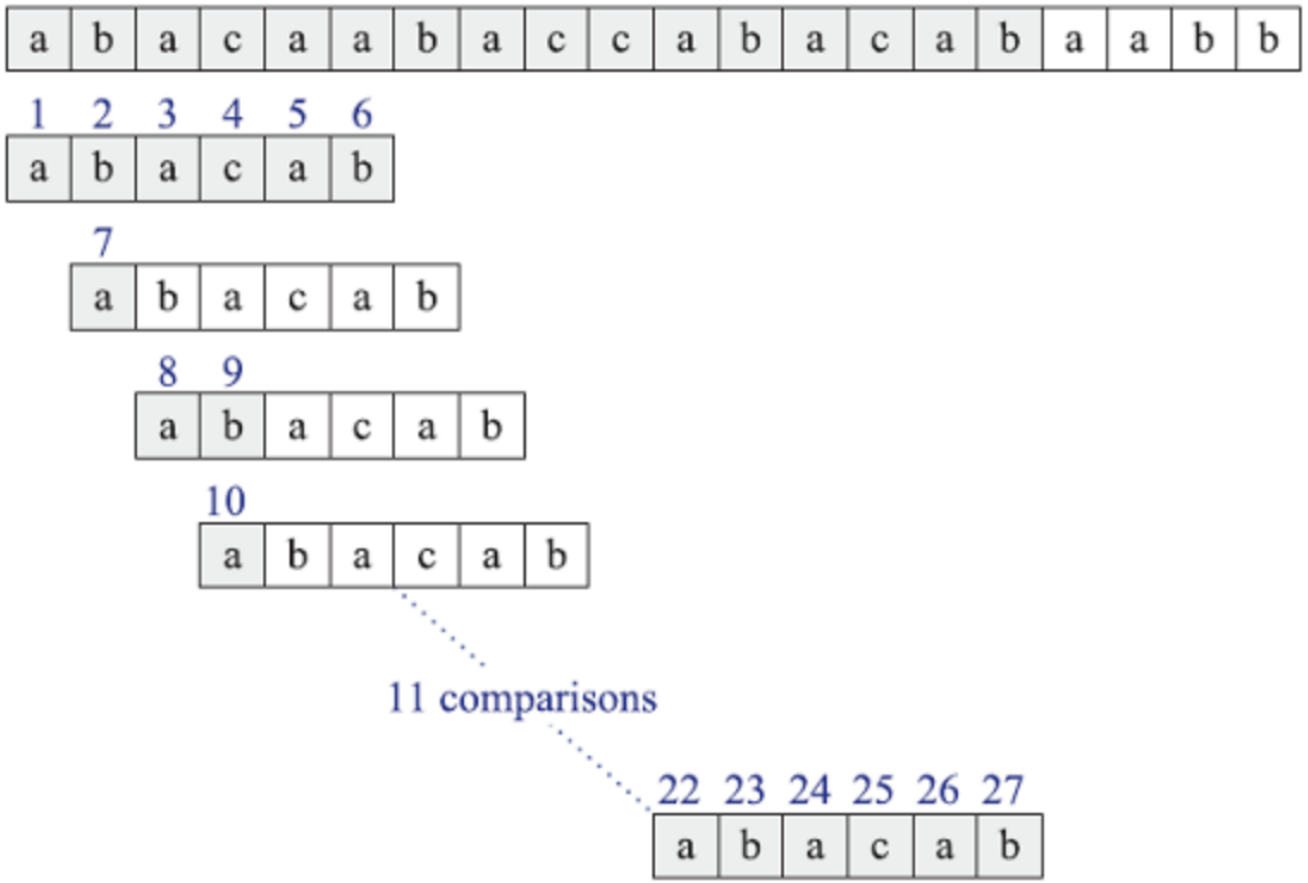
else

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

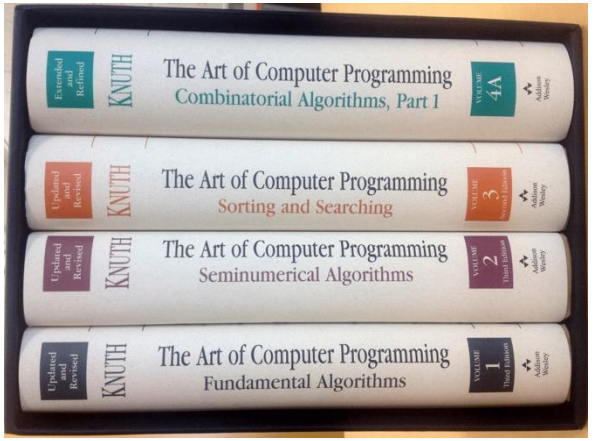
  $1/k + 1/k^2 + 1/k^3 + \ldots + 1/k^m$,

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

Expected-case Analysis for Exact String Matching is **Problematic**

- If a pattern string P and text string T are strings of characters chosen uniformly and independently at random from an alphabet of size k, then the probability that P appears anywhere in T is at most n/k^m.
- For example, if n=1000, m=10, and k=50, then the probability of a match of P in T is about 1 in 10 trillion!
- In this case, a fast (and very accurate) exact matching algorithm is:

Donald Knuth

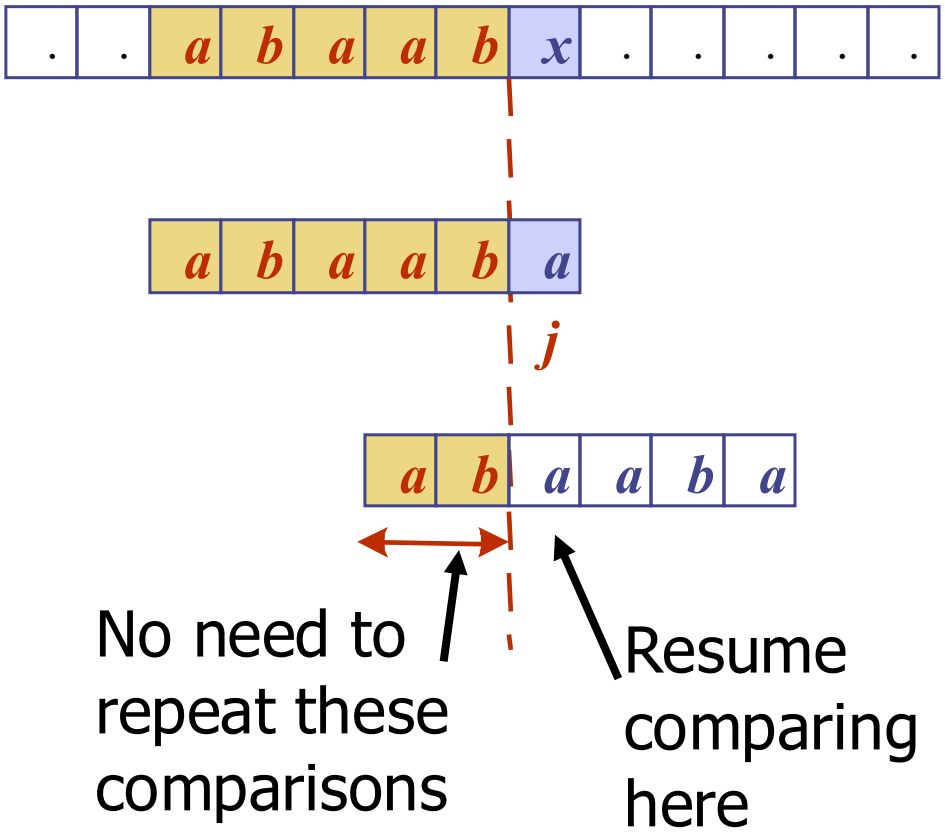


- 1973: Discovered the KMP algorithm (which was also published in a technical report by Morris and Pratt in 1970—all three published a joint paper describing the algorithm in 1977).
- 1974: Received the Turing Award.
- He is also known for his book series, “The Art of Computer Programming,” which formalized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

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

The KMP Algorithm

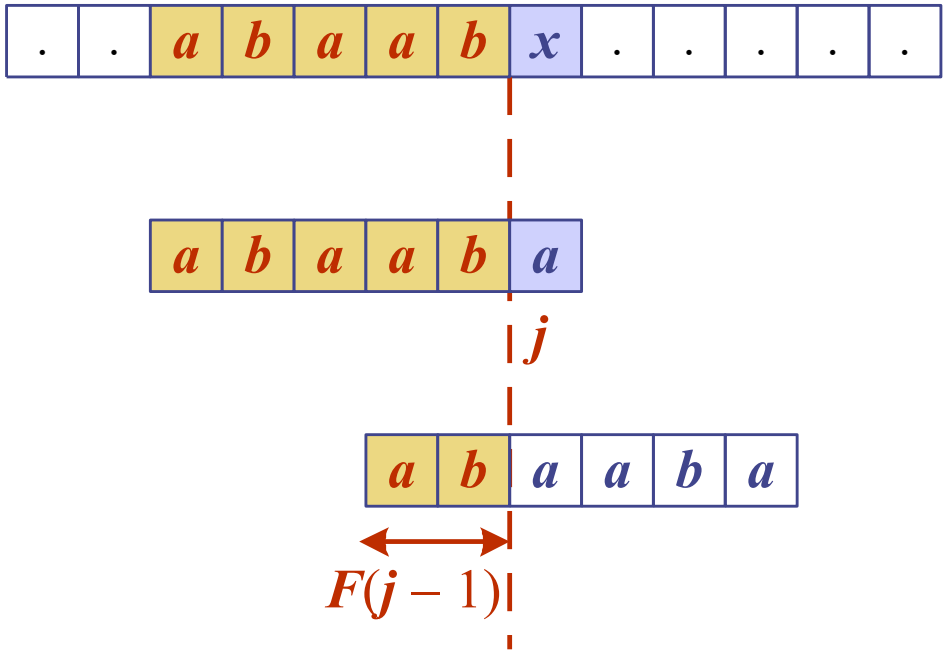
- Consider the comparison of a pattern with a text as in the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the **largest** prefix of $P[0..j]$ that is a suffix of $P[1..j]$
- This approach is similar to the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

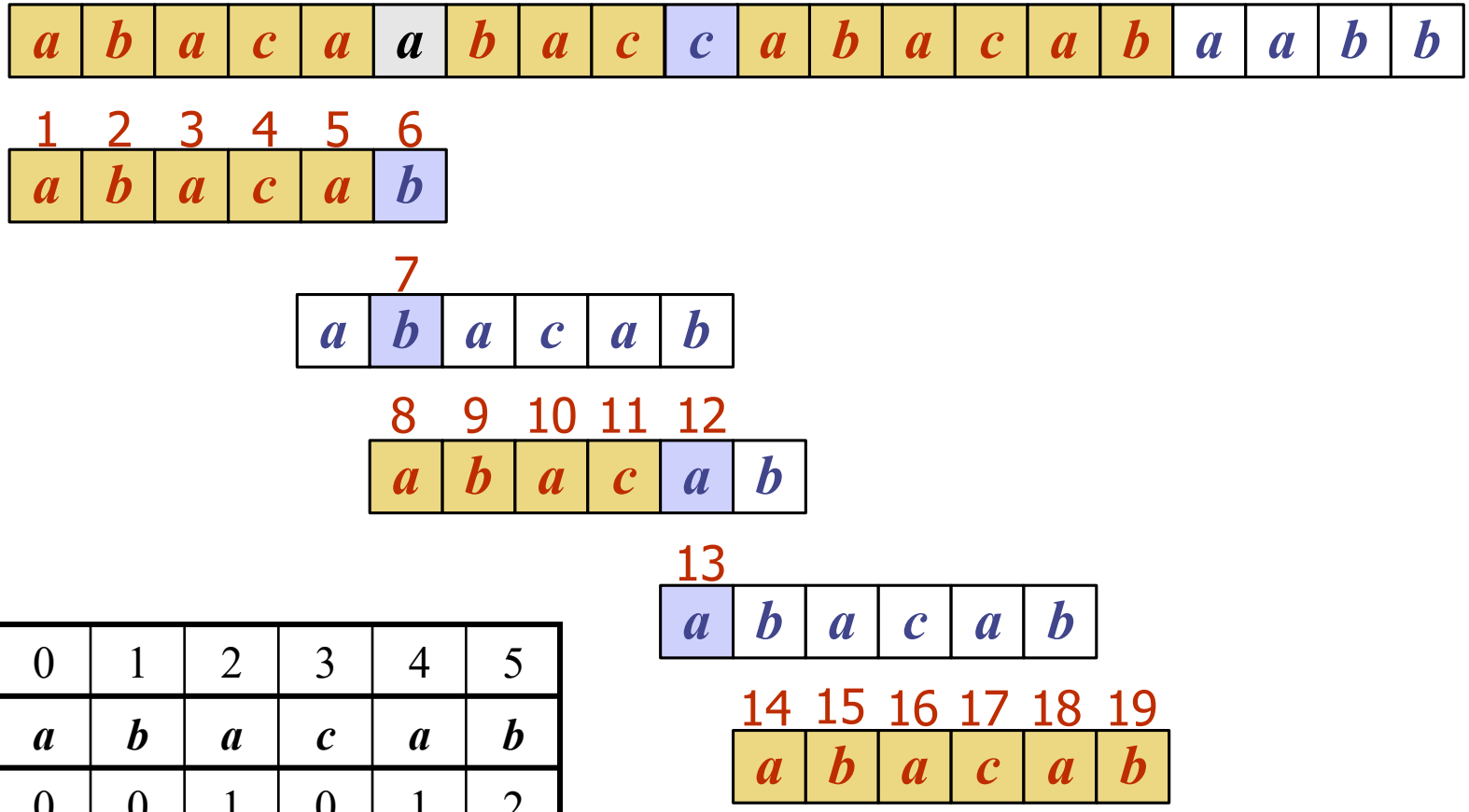
Computing the Failure Function

- 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



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

Summary for KMP

- Thus, the KMP algorithm runs in O(m+n) time for constant-size alphabets.

The Data Type Duality Principle

- 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 often referred to as data type **duality**.

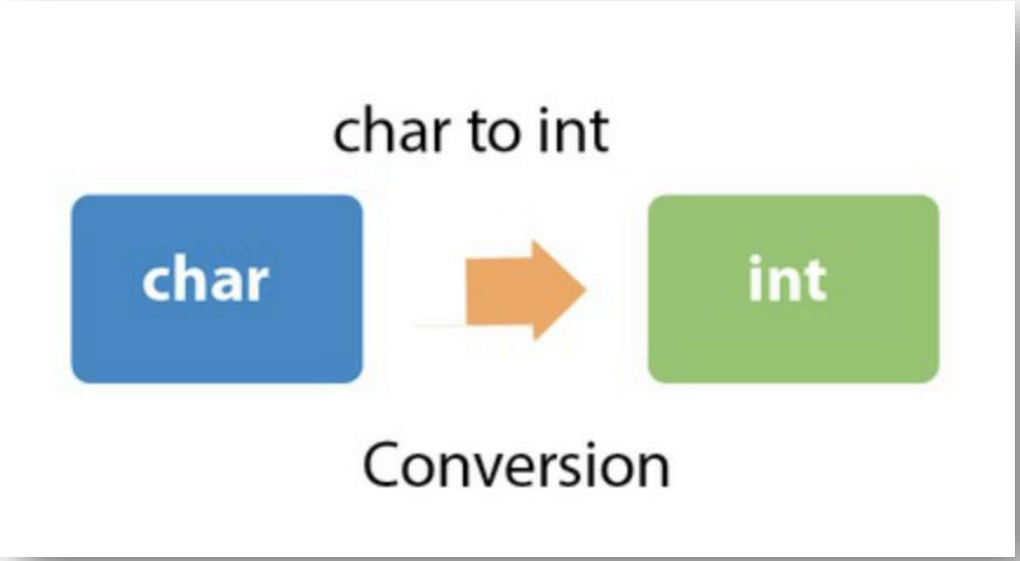
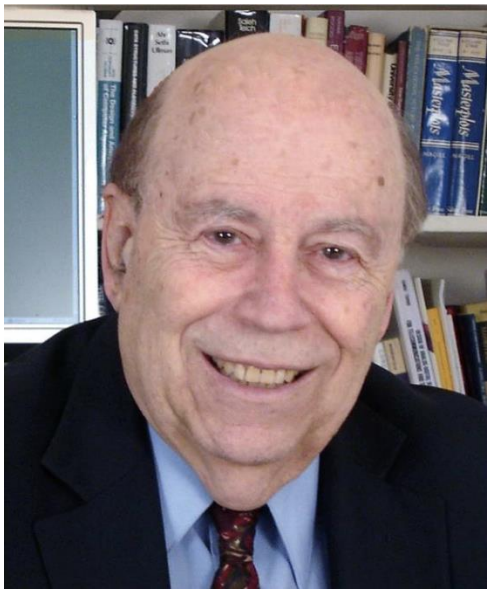


Image from https://www.javatpoint.com/java-char-to-int

Michael Rabin



- 1959: He introduced the nondeterminism concept.
- 1976: He received the Turing Award
  - The “Nobel Prize” of Computer Science

Images from https://en.wikipedia.org/wiki/Michael_O._Rabin, https://en.wikipedia.org/wiki/Dana_Scott

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.
- He was also the PhD advisor to UCI Professor Sandy Irani.

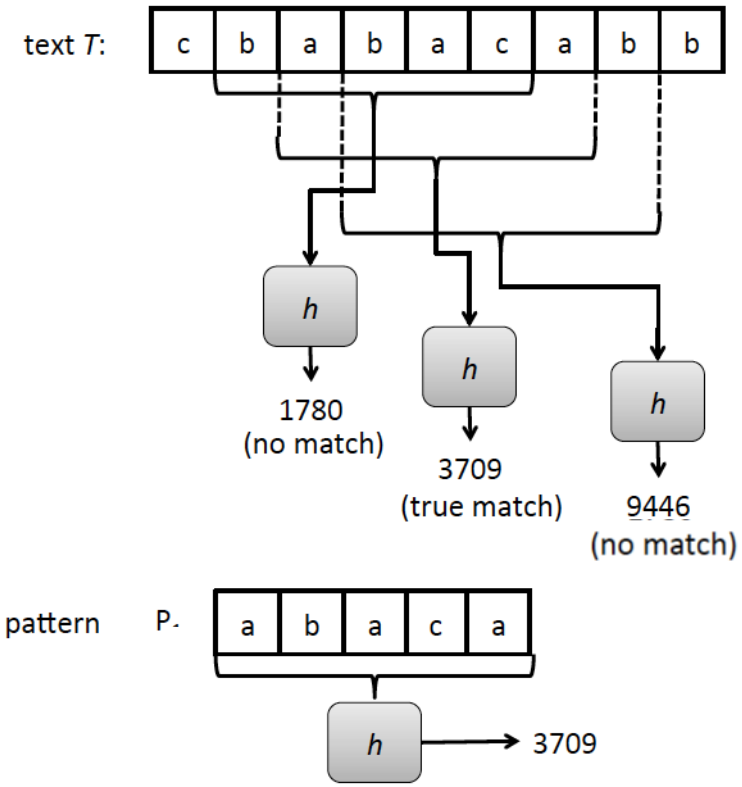
Image from https://en.wikipedia.org/wiki/Richard_M._Karp

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.)

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.

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).

The Multi-Pattern Matching Problem

- Given a text, T, of length n and a set of k patterns, P_1, …, P_k, each of length at most m, find the first (or each) occurrence of a given pattern in T.

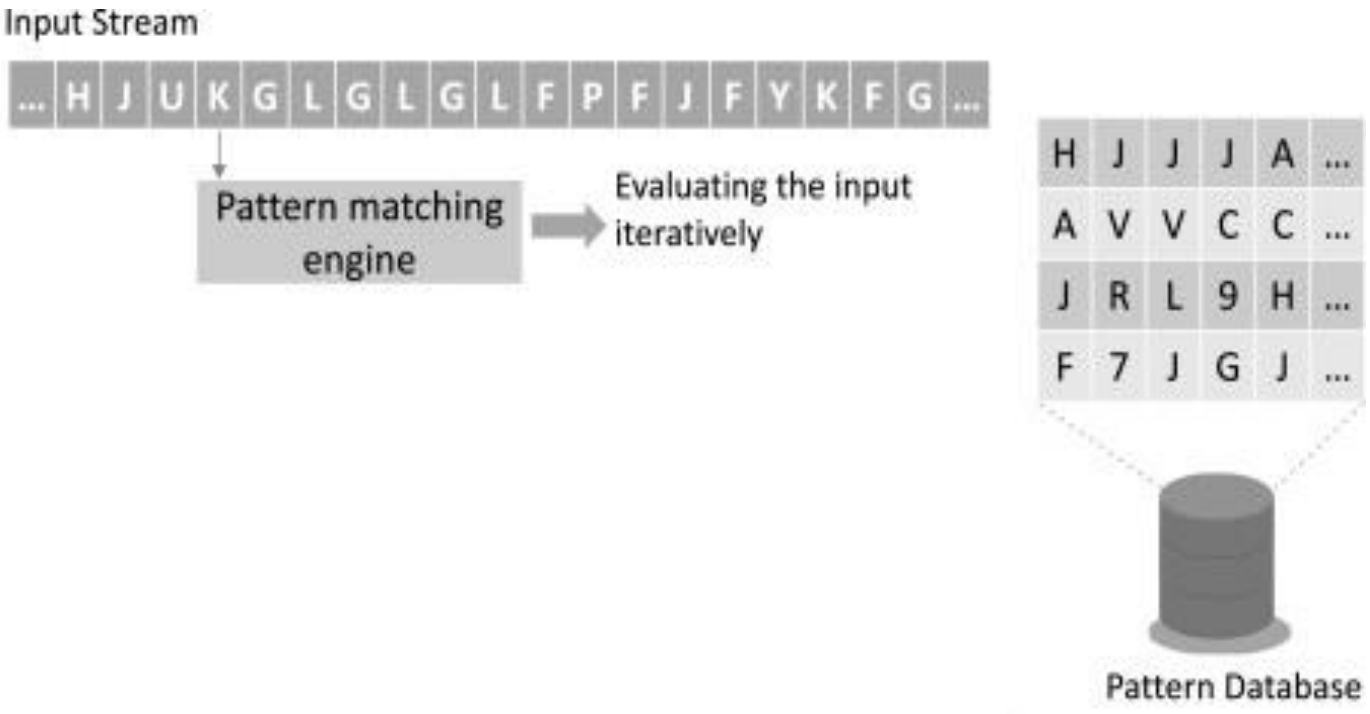


Image from https://www.sciencedirect.com/science/article/abs/pii/S0743731519301984

Multi-Pattern Brute-Force Algorithm

- We can run the brute-force algorithm k times.
- This would run in O(nmk) time, which is very **bad**.

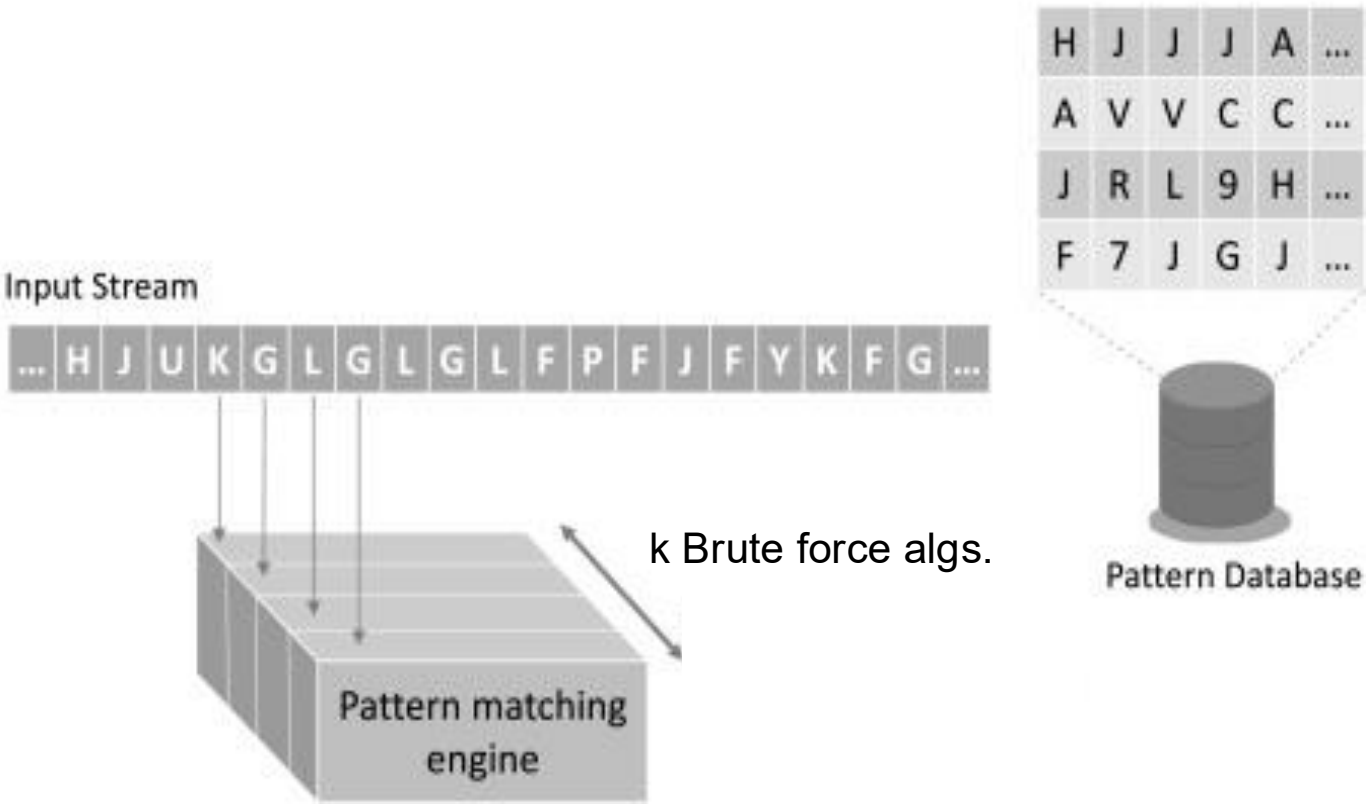


Image from https://www.sciencedirect.com/science/article/abs/pii/S0743731519301984

Alternative: Build a Reg. Exp.

- Given k patterns, P_1, P_2, …, P_k, each of size at most m, build a regular expression:
  - $R = P_1 \mid P_2 \mid \ldots \mid P_k$
- Then, use R to match for any pattern P_i.
- If we simulate the NFA for R, then this takes O(nmk) time.
- This is no better than brute-force!

KMP Algorithm for Multiple Patterns

- We can run the KMP algorithm k times.
- This would run in O((n+m)k) time, which is better than brute force but still not great.

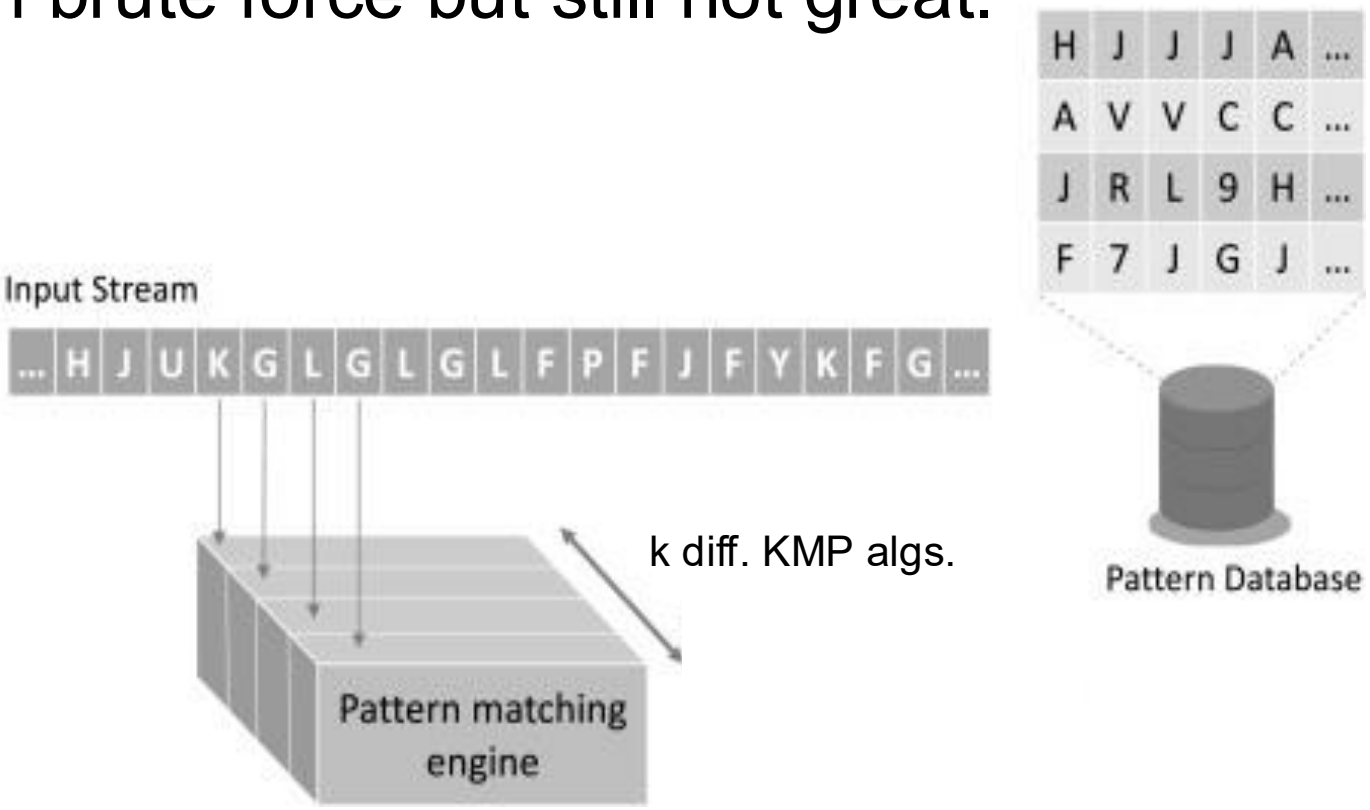


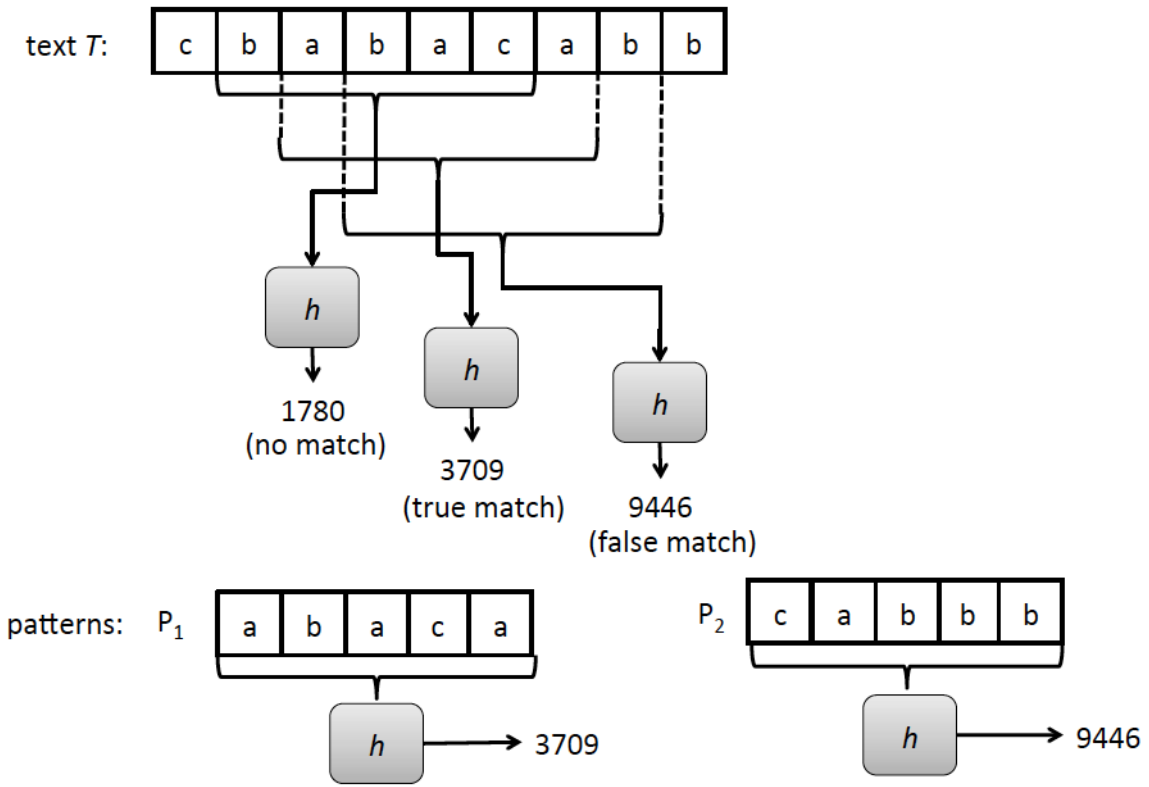
Image from https://www.sciencedirect.com/science/article/abs/pii/S0743731519301984

Recall 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, using a rolling hash function
- 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.

Rabin-Karp for Multiple Patterns

- Text T = cbabacabb
- Pattern P_1 = abaca, P_2 = cabbb



Hash Tables

- The multi-pattern Rabin-Karp algorithm uses hashing in **two** ways:
  1. It uses the hash function, h, for computing the rolling hash values of the pattern and text substrings.
  2. It uses a hash table, H, for storing key-value pairs.
     - This provides expected O(1) time insertions and lookups (see any good reference on hash tables for this)

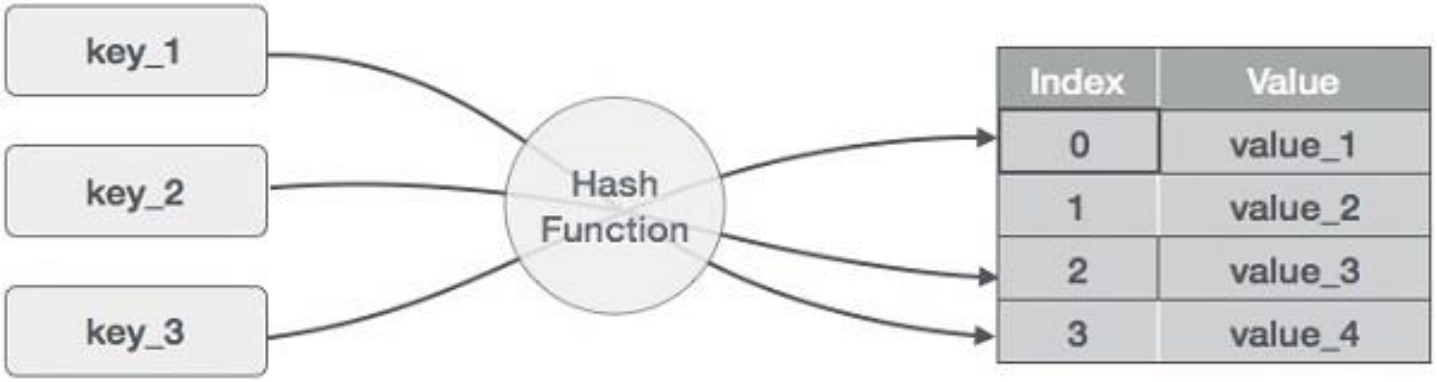


Image from https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm

Rabin-Karp for Multiple Patterns

- Assumes a shiftHash function for computing a shifted hash value.

Input: A set, $L = \{P_1, \ldots, P_l\}$, of l pattern strings, each of length m, and a text string, T, of length n

Output: Each pair, (i, k), such that a pattern, P_k, in L, appears as a substring of T starting at index i

```
1:  Let H be an initially empty set of key-value pairs (with hash-value keys)
2:  Let A be an initially empty list of integer pairs (for found matches)
3:  for k ← 1 to l do
4:      Add the key-value pair, (h(P_k), k), to H
5:  for i ← 0 to n − m do
6:      if i = 0 then   // initial hash
7:          f ← h(T[0..m − 1])
8:      else
9:          f ← shiftHash(f, T, i)
10:     for each key-value pair, (f, k), with key f in H do
11:         // check P_k against T[i..i + m − 1]
12:         j ← 0
13:         while j < m and T[i + j] = P_k[j] do
14:             j ← j + 1
15:         if j = m then
16:             Add (i, k) to A        // a match at index i for P_k
17: return A
```

Analysis of the Rabin-Karp Multi-Pattern Matching Algorithm

- We are given a test of length n, and k patterns, each of length at most m.
- Use a hash function, h, that is random enough so the probability of a false match is at most 1/m.
- Use a hash table, H, that supports expected O(1) time insertions and lookups.
- Building H takes O(km) time, including computing $h(P_i)$ for each pattern, P_i.
- Then the expected running time to find a first match for each pattern (if it exists) is O(n+km).