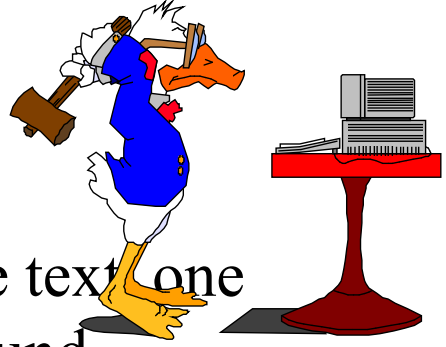
# String Searching

# Brute Force



- The Brute Force algorithm compares the pattern to the text one character at a time, until unmatching characters are found

```
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
-----------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
 ROADS
-----------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
  ROADS
-----------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
   ROADS
-----------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
    ROADS
-----------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
     ROADS
-----------------------------------
```

Total number of comparisons: M (N-M+1)
Worst case time complexity: O(MN)

# Rabin-Karp

- If comparing pattern with text to see if it is the same takes O(1) then brute force would take O(n).
- If we knew the integer value of the pattern and integer value of every substring of text of size m then comparing pattern with text substring is O(1).

Issues:

1. What could be the integer representations of strings.
2. The above assumes that finding integer values of substrings is constant time
3. It assumes that strings are short and will give short integers that fit into registers so that comparison operations on the integers are O(1). What if long integers?

# Rabin-Karp

Issues:

1 What could be the integer representations of strings.

-If only 10 characters then map each character to a digit 0 till 9 and find corresponding decimal value.

-If b types of characters then use base b.

2 The above assumes that finding integer values of substrings is constant time once integer of pattern and first m length substring of text is found

# Issue 2

- 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

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

Intuition: The string '12345' has length 5. The corresponding number would be:
$1.10^4 + 2.10^3 + 3.10^2 + 4.10 + 5$

# Issue 2-continued

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

Given x(i) we can compute x(i+1) for the next subsequence 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

# Issue 3

It assumes that strings are short and will give short integers that fit into registers so that comparison operations on the integers are O(1). What if long integers?

Use a hash function which is mod q for large prime q. This will ensure that integer numbers are small. However, now we don't have the integer to be unique to a substring. Instead we have a hash value which might be same for two dissimilar substrings.

Different hash values mean substrings are different but similar hash value would need a brute force comparison of pattern with that substring.

-What is the hash function used to calculate values for character sequences?

-What is the cost of calculating the hash function each time

-What is the algorithm complexity”

# Rabin-Karp Mods

- If M is large, then the resulting value will be enormous. For this reason, we hash the value by taking it **mod** a prime number **q**.

- Note;

$$[(x \bmod q) + (y \bmod q)] \bmod q = (x+y) \bmod q$$

$$(x \bmod q) \bmod q = x \bmod q$$

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

Now:

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

Earlier:

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

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

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

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

Now: $h(i+1) = ( h(i) \cdot b \bmod q$

$-t[i] \cdot b^{M} \bmod q$

$+t[i+M] \bmod q ) \bmod q$

# Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence 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.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

.

# The Knuth-Morris-Pratt Algorithm

# The KMP Algorithm

- Keeps track of information gained from previous comparisons.

  T: a b x a b x a b b x c y b ……
- P: a b x a b b x c y

- Find the largest text suffix matched so far that is same as pattern prefix

- Find the longest prefix of the pattern P[0,..,j] that is also a suffix of P[1,..,j]

- For each index position of pattern from 1 to j, compute the prefix index position from which one can continue checking.

# Prefix array content

0 1 2 ….k …….m …. N-1

n is the total pattern size,

m is the index that we are trying to fill,

Say longest prefix that matches with the suffix ending at m has length k from 0…k-1 then P[m]=k. Such a k indicates

That if a mismatch occurs on comparing P[m+1] with text location ti then compare ti against P[m] which is k.

# The KMP Algorithm

- For P[m]; m>=1
- compute the largest prefix index k s.t P[0…k-1] overlaps with [m-k+1.….m]
- if no overlap then mark as 0

  0 1 2 3 4 5 6 7
  A b c d a b c a
- 0 0 0 0 1 2 3 1

  Fill in the values
  A a b a a b a a a

# The KMP Algorithm

- For each index position of patter from 1 to j, compute the prefix index position from which one can continue checking.

I:0 1 2 3 4 5 6 7 8
a a b a a b a a a
V:0 1 0 1 2 3 4 5 2

0 1 2 3 4 5
a b a c a b
0 0 1 0 1 2

# Prefix algorithm intuition

We are trying to fill the value of A[i] which says that if till the ith character the pattern and text match and P[i+1] does not match with text then the mismatched text character should be compared with P[A[i]]. So A[i] gives index with which I should compare text character.

How to fill A[i] ?

I need to know the longest suffix until i that matches with prefix. The longest suffix until i-1 that matches with some prefix (say P[0…..j-1]) is P[i-j……i-1].

1. if P[i]=P[j] then the longest suffix until i to match prefix is P[0….j]. Hence A[i]=j+1

# Prefix algorithm intuition

How to fill A[i] ?

I need to know the longest suffix until i that matches with prefix. The longest suffix until i-1 that matches with some prefix (say P[0…..j-1]) is P[i-j……i-1].

2. if P[i]!=P[j], then the longest suffix till i might be the second longest suffix till i-1 with P[i] if the next character of prefix is same as P[i]. Else might be the third longest suffix till i-1 with P[i] if the prefix's next character is same as p[i] and so on.

How do we find the next longest suffix until i-1 after each failed attempt to see if the next character of the next longest prefix is same as P[i]]

# Prefix algorithm intuition

How do we find the next longest suffix until i-1 after each failed attempt to see if the next character of the next longest prefix is same as P[i]]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | a | b | c | d | a | b | c | a | b | c | a |
| 0 | 0 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 4 |

Suffix till i-1 is same as suffix of P[i-j,…..i-1] that is prefix same as suffix of P[0,…..j-1] that is a prefix.

A[j-1] gives you the index right after the suffix of P[0,…..j-1] matches with prefix. If P[A[j-1]]!=P[i] then

We need to look for suffix of P[0,…..j-1] which is prefix.

Same as look for suffix of P[0,……., (A[j-1]-1)] which is prefix and so on.

# The KMP Algorithm (contd.)

- the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch(T,P)

**Input**: Strings T (text) with n characters and P (pattern) with m characters.

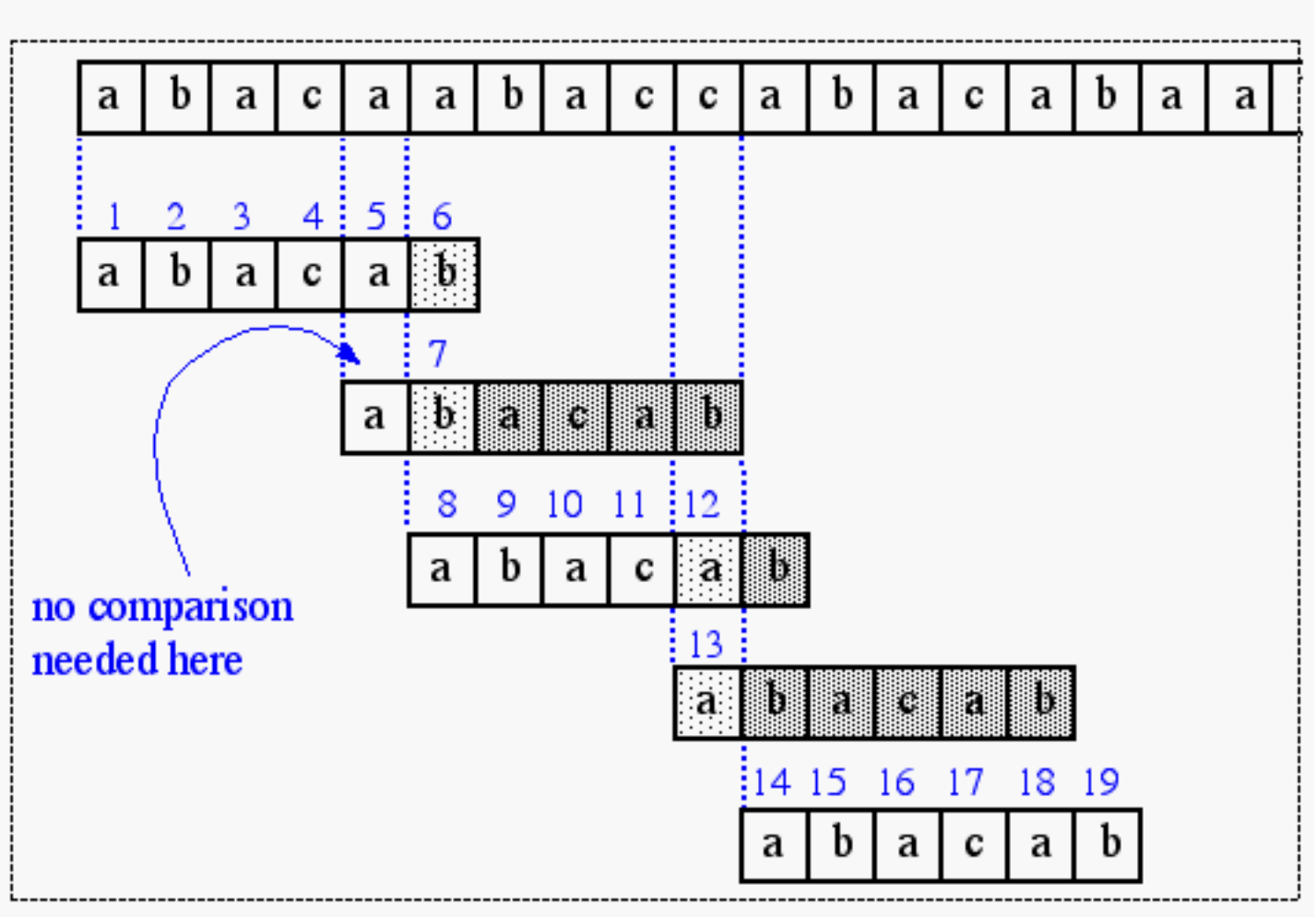
**Output**: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

# Algorithm

- ```
  vector<int> PrefixFunction(string S)
  {
   vector<int> p(S.size());
   int j = 0; p[0]=0;
   for (int i = 1; i < (int)S.size(); i++)
   { while (j > 0 && S[j] != S[i])
           j = p[j-1];
     if (S[j] == S[i]) j++;
     p[i] = j;
    }
   return p;
  }
  ```
  ```

a b a c a b
1 1 2 1 2 3

Note:indices starting from 1



# Algorithm

```
f ← KMPFailureFunction(P)
    i ← 0
    j ← 0
    while i < n do
        if P[j] = T[i] then
            if j = m - 1 then
                return i - m - 1 {a match}
            i ← i + 1
            j ← j + 1
        else if j > 0 then {no match, but we have advanced}
            j ← f(j-1) {j indexes just after matching prefix in P}
        else
            i ← i + 1
    return "There is no substring of T matching P"
```

# The KMP complexity

- The inner while loop of the preprocessing algorithm decreases the value of $j$ by at least 1, since $b[j] < j$. The loop terminates at the latest when $j = 0$, therefore it can decrease the value of $j$ at most as often as it has been increased previously by j++. Since j++ is executed in the outer loop exactly $m$ times, the overall number of executions of the inner while loop is limited to $m$. The preprocessing algorithm therefore requires $O(m)$ steps.

- From similar arguments it follows that the searching algorithm requires $O(n)$ steps. The above example illustrates this: the comparisons (green and red symbols) form "stairs". The whole staircase is at most as wide as it is high, therefore at most $2n$ comparisons are performed.
- Since $mn$ the overall complexity of the Knuth-Morris-Pratt algorithm is in $O(n)$.

# Boyer Moore Algorithm

- Three ideas: Each idea can be used independently

--Scan right to left of pattern

--Bad character rule

--Good suffix rule

Using three together gives better results.

# Idea1: right to left scan

T :abcbbdbdebre

P :hearih

Compare from last character of pattern towards left

Naive: For a mismatch shift pattern to right by one jump.

# Idea2:Bad Character rule

T: adjaiutnansk
P: kisudu

T: adjaiutnansk
P: kasudu

--Allign the mismatched character in T (i in T here) with right most occurrence of i to the left of d in P
--If no such occurrence then shift by length of pattern

T: adjaiutnansk
P: kisudu

T: adjaiutnansk
P: kisudu

# Idea 3(a): good suffix rule

Case1: matched portion exitst to left of matched area not preceded by the mismatched character.

T: wadssakdcabcajajst

P: dabccabcaabc

--Allign matched part of text with closest left occurrence of matched pattern which is not preceded by mismatch text character.

--If no match go to Case2.

T: wadssakdcabcajajst

P: dabccabcaabc

# Idea 3(b): good suffix rule

Case2 : match portion does not exist to left of matched area

T: wadssakdcabcajajst

P: bcccakcaabc

--Find longest prefix of P that matches with suffix of P.

T: wadssakdcabcajajst

P: bccccakcaabc

---If no such prefix then shifted by length of pattern.

# Precomputations required

Bad Character Rule:

For each character find its previous occurrence.

Good suffix Rule:

Case1: Find closest left occurrence of matched pattern not preceded by mismatch character

Case2: Compute longest prefix that is suffix of pattern

# Precomputations required

Bad Character Rule: For mismatch text character c, find its occurrence in pattern.

--For alphabets in pattern, store rightmost occurrence of each character in R(c).
--Let index denote the index of pattern character that mismatched the text character c

Shift =index-R(c) if R(c)< index
=1 if R(c)> index

# Bad Character rule precomputation (m-1-index)

T: adjaiutnansk
P: kisudu

012345
kisudu

Characters: k i s u d
0 1 2 5 4

T: adjaiutnsnsk
P: kisudu

T: adjaiutnsnsk
P: kisudu

# Precomputations required

Good suffix Rule:

Case1: Find closest left occurrence of matched pattern not preceded by mismatch character

Homework ☺

How to compute case1 of good suffix rule

What would be the fastest string search algorithm

# Which idea to use?

Each idea or rule is complete where each step is independent of the previous.

Hence at each step use the rule that gives maximum shift.

# Boyer Moore Horspool

Horspool proposed to use only the bad-character shift of the rightmost character of the window to compute the shifts in the Boyer-Moore algorithm.

The bad-character shift used in BM algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful.