String Matching

History

- 1970 S.Cook proved that there exists an $O(m+n)$ algorithm. No construction!

- 1976 Knuth-Morris-Pratt (KMP) provided an $O(n+m)$ algorithm

- 1976 Boyer-Moore produced a sub-linear algorithm

- 1980 Karp-Rabin :
  very simple
  good average case
  $O(m \cdot n)$ in worst case

- better methods were found

The problem

<u>Given:</u> text string $T[1 \ldots n]$ and pattern string $P[1 \ldots m]$, find pattern in text.

FIND shift s, $0 \leq s \leq n - m$, such that:

$$T[s+1 \ldots s+m] = P[1 \ldots m]$$

$$\begin{array}{cccccccccc} t_1 & t_2 & \cdots & t_{s+1} & t_{s+2} & \cdots & t_{s+m-1} & t_{s+m} & \cdots & t_n \\ & & & p_1 & p_2 & \cdots & p_{m-1} & p_m & & \end{array}$$

Or, equivalently, find matching position i, $1 \leq i \leq n - m + 1$, such that:

$$T[i \ldots i+m-1] = P[1 \ldots m]$$

$$\begin{array}{cccccccccc} t_1 & t_2 & \cdots & t_i & t_{i+1} & \cdots & t_{i+m-1} & t_{i+m} & \cdots & t_n \\ & & & p_1 & p_2 & \cdots & p_m & & & \end{array}$$

The naive string-matching algorithm:

Naive-String-Matcher(T, P)

$n =$*Length*(T)
$m =$*Length*(P)
for $s := 0$ *to* $n - m$ *do*
    if $T[s + 1 \ldots s + m] = P[1 \ldots m]$ *then*
        print ("pattern found with shift s")

Time for naive-string-matcher

$O(n \cdot m)$

The reason for the inefficiency:
The Back-up for each Mismatch!

Example:
$P = aaab$
$T = aaaaaaaa \ldots aaaab$

For each position from 1 to $n - 4$ we need four comparisons to find out the mismatch.

For position $n - 3$, we need four comparisons to find the match.

Total time $= (n - 4) \cdot 4 = (n - m)m \approx nm$

String matching with finite state automata

- Given pattern P, it is possible to construct a finite state automaton that can be used to scan the text T for a copy of P very quickly.
- finite automaton is a special kind of machine or flowchart
- Σ: alphabet of strings, $\alpha = |\Sigma|$

The flowchart has three types of **nodes**:

– start node

– stop/accept node:

pattern found

– read node:

read next character in text

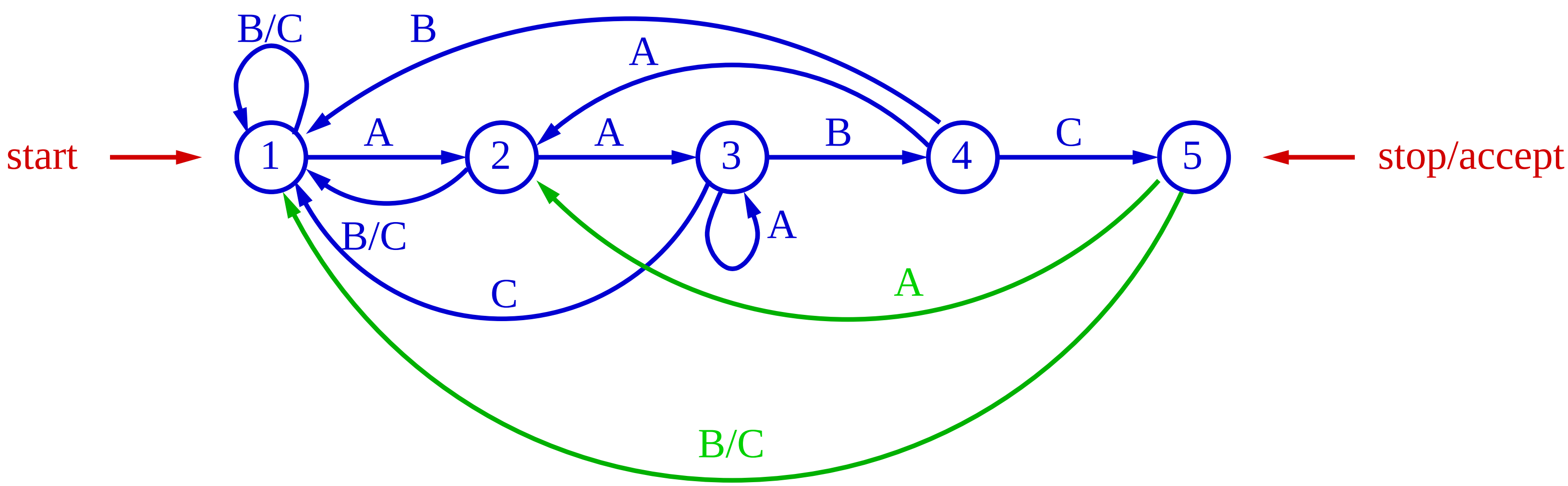
if no more characters, halt, no match

Arcs

– α logical arcs from each read node.

Example

Pattern: $P = AABC$



node 1: there is no partial match

node 2: match A, node 3: match AA, node 4: match AAB

node 5: match the pattern

Text: $T = CCAACCAABAABCA$

Can scan the text in $O(n)$ time!!

Main idea: do not back up when scanning the text.

Problem with Finite Automata:

- too many arcs
- for each node, we have $|\alpha|$ arcs!

KMP

- Avoid back-up
- on mismatches, don't back-up

We do not decrease the pointer to the text.
Instead, we slide the pattern to the right (try next *possible* position, not every position).

Example:

		$\downarrow$					$\downarrow$							
T:	C	A	B	A	B	A	B	A	B	C	B	C		
		$\|$	$\|$	$\|$	$\|$	$\|$	$\|$	$\Updownarrow$						
P:		A	B	A	B	A	B	C						
			A	B	A	B	A	B	C					$\leftarrow$ impossible
				A	B	A	B	A	B	C				$\leftarrow$ possible
					A	B	A	B	A	B	C			$\leftarrow$ impossible
						A	B	A	B	A	B	C		$\leftarrow$ possible
							A	B	A	B	A	B	C	$\leftarrow$ impossible

How far to the right should we slide pattern?

As far as possible (to save comparisons) without missing potential matches.

Example:

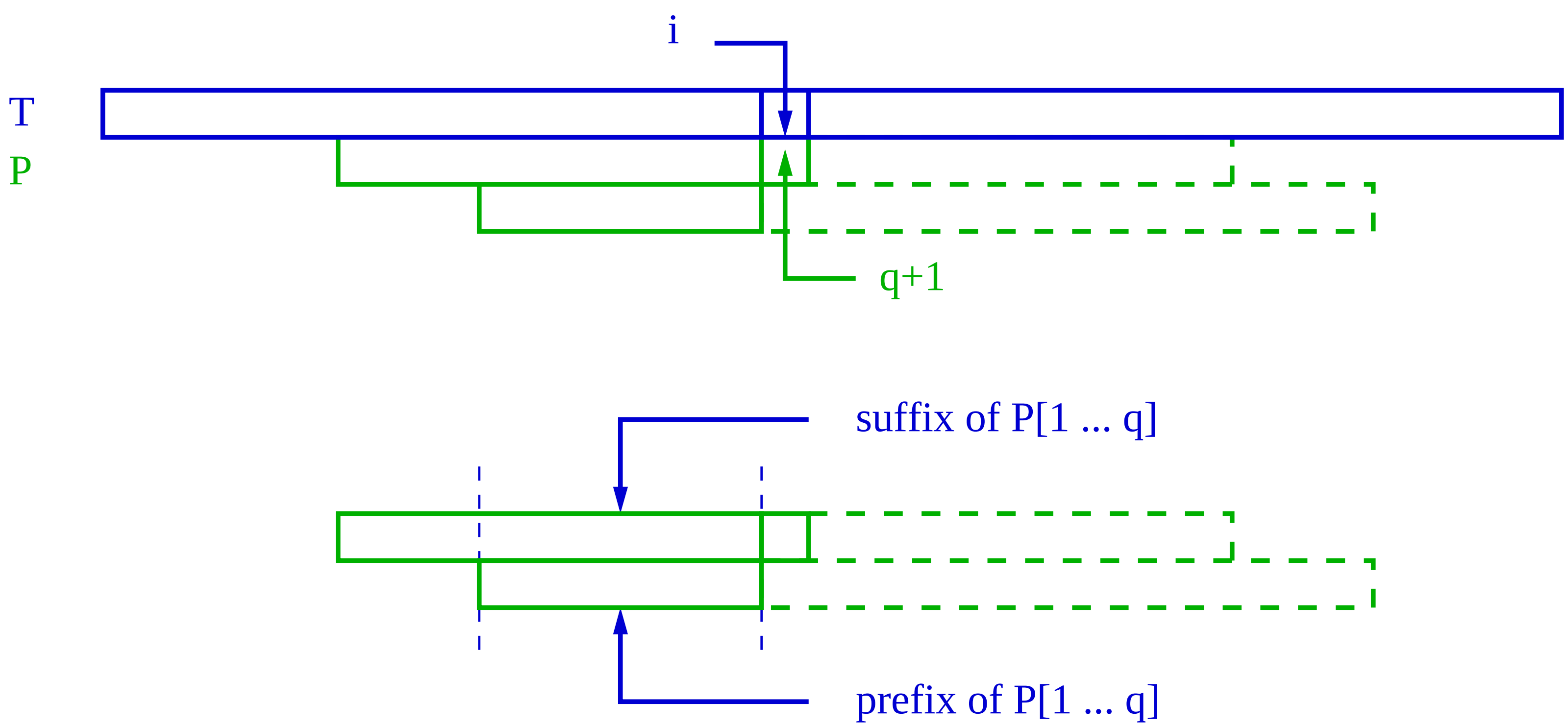
$$
\begin{array}{lcccccccccccccl}
 & & \downarrow & & & & & & \downarrow & & & & & & \\
\text{T:} & \text{C} & \text{A} & \text{B} & \text{A} & \text{B} & \text{A} & \text{B} & \text{A} & \text{B} & \text{C} & \text{B} & \text{C} & & \\
 & & \| & \| & \| & \| & \| & \| & \Updownarrow & & & & & & \\
\text{P:} & & \text{A} & \text{B} & \text{A} & \text{B} & \text{A} & \text{B} & \text{C} & & & & & & \\
 & & & & \text{A} & \text{B} & \text{A} & \text{B} & \text{A} & \text{B} & \text{C} & & & & \\
 & & & & & & \text{A} & \text{B} & \text{A} & \text{B} & \text{A} & \text{B} & \text{C} & & \leftarrow \text{moving too far}
\end{array}
$$

Suppose that we have matched the first q characters of P and have a mismatch at $T[i]$ and $P[q+1]$.

- this means that $T[i-q \ldots i-1] = P[1 \ldots q]$.
- current matching position is $i-q$.
- naive algorithm will try position $i-q+1 = i-(q-1)$ by setting $i = i-q+1$ and $q = 0$.
- we want to keep i unchanged and try some position $i-q_1$, where $q_1 < q$.
- this means that we will compare $T[i]$ with $P[q_1+1]$.
- this can only be useful if $T[i-q_1 \ldots i-1] = P[1 \ldots q_1]$.
- since $T[i-q \ldots i-1] = P[1 \ldots q]$ and $q_1 < q$, this means that $P[q-q_1+1 \ldots q] = P[1 \ldots q_1]$.
- if there are more than one such positions of q_1, then we should try the largest one to avoid missing potential matching positions.
- in the algorithm, we do not need to explicitly maintain the matching position $i-q$, instead we keep i, the pointer to the text, and q, the number of characters matched.

More precisely...

If at some point we had a mismatch at $P[q+1]$ we want to find the *largest proper prefix* of $P[1 \ldots q]$ that is equal to a *suffix* of $P[1 \ldots q]$



Why largest prefix? We do not want to miss potential matches!

Important observation!

The information we need to determine how far we can slide pattern is only dependent on the pattern!

We can do a preprocessing for pattern!

Informal Description of the algorithm

- Text T is always scanned forward.
- There is no backtracking in T, although the same character of T may compare to several characters of pattern P. (When there are mismatches.)
- When a mismatch is found, we consult a table, based on how many characters are currently matched, to determine the next character in P that should be compared to current character in T.
- There is an entry for each location, q, in P which tells us when a mismatch occurs at $P[q+1]$, how we should determine the next character $P[q_1+1]$, $q_1 < q$, that would be compared with current character in T.

The table

We call this table *next*. (In our textbook: prefix function or π)

$next(i)$ = the maximum j $(0 < j < i)$ such that $b_1 b_2 \ldots b_j = b_{i-j+1} b_{i-j+2} \ldots b_i$

$next(i) = 0$ if no such j exists.

Therefore we have $next(1) = 0$ (since there is no j such that $0 < j < 1$)

Example: P= ABABC

$$\begin{array}{rcccccc} i & = & 1 & 2 & 3 & 4 & 5 \\ P & = & A & B & A & B & C \\ next & = & 0 & 0 & 1 & 2 & 0 \end{array}$$

We will consider how to compute next table later.
It can be constructed in $O(m)$ time!

The matching process

(Assume we have already computed next table.)

- Characters in text T are compared to characters in pattern P *until there is a mismatch.*
- Now, say $P[q+1]$ mismatches, the next table is consulted. *The same character in T is compared to $P[next[q]+1]$.*

  (Since the first $next[q]$ characters already match, keep pointer to T unchanged, change the pointer to P from $q+1$ to $next[q]+1$.)
- If this is a mismatch too, then compare to $P[next[next[q]]+1]$ and so on.
- only exception: when mismatch is against $P[1]$ $(q=0)$. In this case, we increase the pointer to T by one.

Algorithm: String_Matching(T, n, P, m)
Input: text $T[1 \ldots n]$ and pattern $P[1 \ldots m]$.
Output: matching positions.

```
begin
    i := 1;
    q := 0;
    while i ≤ n do
        if T[i] == P[q+1] then
            i := i+1;
            q := q+1;
        else
            if q == 0 then
                i := i+1;
            else
                q := next[q];
        if q == m then
            print "pattern found at position" i-m;
            q = next[q];
end
```

Complexity

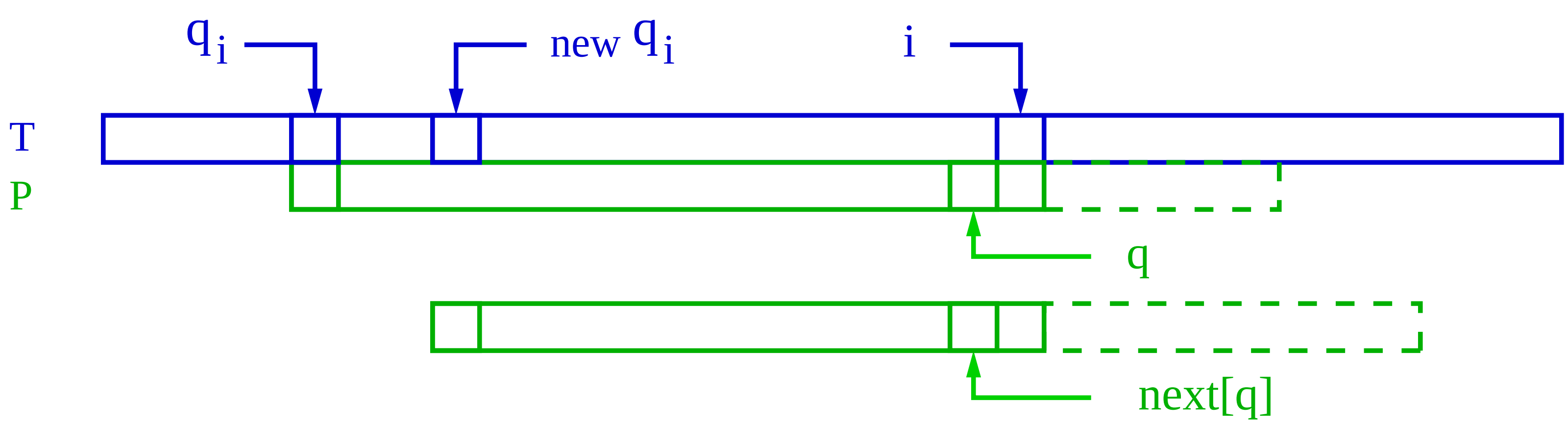
For matching process

– i is the current pointer to text T.

It either is increased by 1 or remains unchanged.

– Let $q_i = i - q$.

q_i is the current matching position of pattern P in text T.



– When i remains unchanged, q_i is increased.

– When q_i remains unchanged, i is increased.

– In any step either i is increased or q_i is increased.

– The running time will be less than or equal to the *the number of increments of* i *plus the number of increments of* q_i.

– Since when the algorithm terminates,

$i \leq n + 1$ and $q_i \leq n$,

Time $\leq$ total increments $\leq 2n$ which implies $O(n)$

Compute table $next[\,]$

By induction

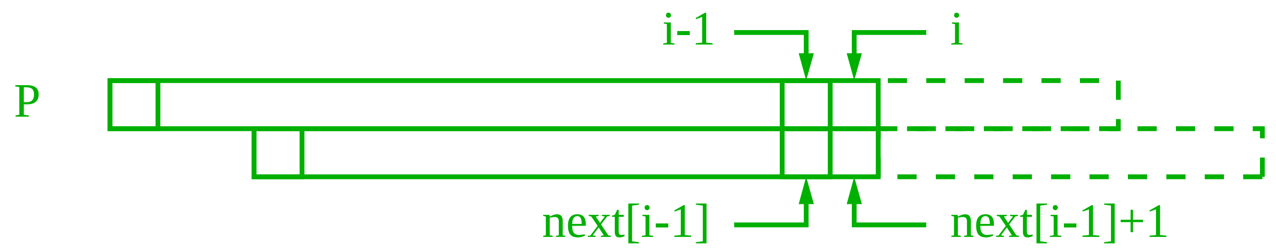
- Base case: $next[1] = 0$.
- Assume we have computed $next$ for $1, 2, \ldots i-1$.
- Consider $next[i]$.

  † $next[i] \le next[i-1]+1$

  † If $P[i] = P[next[i-1]+1]$ then
  $next[i] = next[i-1]+1$ (best possible for $next[i]$).

  † If $P[i] \neq P[next[i-1]+1]$ then
  compare $P[i]$ to $P[next[next[i-1]]+1]$ and so on. (exactly the same as we have a mismatch in matching process.)

  † Continue. Either we find a match, or there is no match and $next[i] = 0$.



For computing next table

Algorithm: Compute_Next(P, m)
Input: pattern $P[1 \ldots m]$.
Output: next table (an array of size m).

begin
    $\textit{next}[1] = \textit{0;}$
    for $i := 2$ *to* m *do*
        $q := \textit{next}[i-1];$
        while $q > 0$ *and* $P[i] \neq P[q+1]$ *do*
            $q := \textit{next}(q);$
        if $P[i] == P[q+1]$ *then*
            $q := q + 1;$
        $\textit{next}[i] := q;$
end

For computing next table

We modify the program to the following:

$begin$

$\quad next[1] = 0;$

$\quad \underline{q := 0;}$

$\quad for\ i := 2\ to\ m\ do$

$\quad\quad while\ q > 0\ and\ P[i] \neq P[q+1]\ do$

$\quad\quad\quad q := next(q);$

$\quad\quad if\ P[i] == P[q+1]\ then$

$\quad\quad\quad q := q + 1;$

$\quad\quad next[i] := q;$

end

Again, let $q_i = i - q$

Either i is increased or q_i is increased.

When i remains unchanged, q_i is increased.

When q_i remains unchanged, i is increased.

When the algorithm terminates, $i \leq m$ and $q_i \leq m$.

Time $\leq$ total increments $\leq 2m$ which implies $O(m)$