Today we're going to cover

- String matching
  - Naive algorithm
  - Knuth–Morris–Pratt (KMP) algorithm
- Tries
- Suffix tries
- Suffix trees
- Suffix arrays

String problems

- Strings frequently appear in our kind of problems
  - Reading input
  - Writing output
  - Parsing
  - Identifiers/names
  - Data
- But sometimes strings play the key role
  - We want to find properties of some given strings
  - Is the string a palindrome?
- Here we're going to talk about things related to the latter type of problems
- These problems can be hard, because the length of the strings are often huge

String matching

- Given a string S of length n,
- and a string T of length m,
- find all occurrences of T in S

- Note:
  - Occurrences may overlap
  - Assume strings contain characters from a constant-sized alphabet

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$

- Three occurrences:

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$

- Three occurrences:
  - $\mathrm{cabc}\mathrm{aba}\mathrm{bacaba}$

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$

- Three occurrences:
  - $\text{cabc}\textcolor{red}{\text{aba}}\text{bacaba}$
  - $\text{cabcab}\textcolor{red}{\text{aba}}\text{caba}$

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$

- Three occurrences:
  - cabcababacaba
  - cabcababacaba
  - cabcababacaba

Naive string matching algorithm

- For each substring of length m in S,
- check if that substring is equal to T.

Naive string matching algorithm

- S: bacbababaabcbab
- T: ababaca

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `  ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `  ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: bacbababaabcbab
- T: ababaca

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: bacbababaabcbab
- T: ababaca

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

```
int string_match(const string &s, const string &t) {
    int n = s.size(),
        m = t.size();

    for (int i = 0; i + m - 1 < n; i++) {
        bool found = true;
        for (int j = 0; j < m; j++) {
            if (s[i + j] != t[j]) {
                found = false;
                break;
            }
        }
        if (found) {
            return i;
        }
    }

    return -1;
}
```

Naive string matching algorithm

- Double for-loop
  - outer loop is $O(n)$ iterations
  - inner loop is $O(m)$ iterations worst case
- Time complexity is $O(nm)$ worst case

Naive string matching algorithm

- Double for-loop
  - outer loop is $O(n)$ iterations
  - inner loop is $O(m)$ iterations worst case
- Time complexity is $O(nm)$ worst case
- Can we do better?

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: bacbababaabcbab
  - T: ababaca

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `  ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `  ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: bacbababaabcbab
  - T: ababaca

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`
- The number of shifts depend on which characters are currently matched

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$
- Example:

i	1	2	3	4	5	6	7
$T[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$
- Example:

i	1	2	3	4	5	6	7
$T[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

- If, at position i, q characters match (i.e. $T[1 \ldots q] = S[i \ldots i + q - 1]$), then
  - if $q = 0$, shift pattern 1 position right
  - otherwise, shift pattern $q - \pi[q]$ positions right

Knuth–Morris–Pratt algorithm

- Example:
  - S: bacbababaabcbab
  - T: ababaca

Knuth–Morris–Pratt algorithm

- Example:
  - S: bacbababaabcbab
  - T: ababaca
  - 5 characters match, so $q = 5$

Knuth–Morris–Pratt algorithm

- Example:
  - S: bacbababaabcbab
  - T: ababaca
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$
  - Then shift $q - \pi[q] = 5 - 3 = 2$ positions

Knuth–Morris–Pratt algorithm

- Example:
  - S: bacbababaabcbab
  - T: ababaca
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$
  - Then shift $q - \pi[q] = 5 - 3 = 2$ positions
  - S: bacbabababaabcbab
  - T: ababaca

Knuth–Morris–Pratt algorithm

- Given π, matching only takes $O(n)$ time
- π can be computed in $O(m)$ time
- Total time complexity of KMP therefore $O(n + m)$ worst case

Knuth–Morris–Pratt algorithm

```
int* compute_pi(const string &t) {

    int m = t.size();
    int *pi = new int[m + 1];
    if (0 <= m) pi[0] = 0;
    if (1 <= m) pi[1] = 0;
    for (int i = 2; i <= m; i++) {
        for (int j = pi[i - 1]; ; j = pi[j]) {
            if (t[j] == t[i - 1]) {
                pi[i] = j + 1;
                break;
            }
            if (j == 0) {
                pi[i] = 0;
                break;
            }
        }
    }

    return pi;
}
```

Knuth–Morris–Pratt algorithm

```
int string_match(const string &s, const string &t) {

    int n = s.size(),
        m = t.size();

    int *pi = compute_pi(t);

    for (int i = 0, j = 0; i < n; ) {
        if (s[i] == t[j]) {
            i++; j++;
            if (j == m) {
                return i - m;
            }
        }
        else if (j > 0) j = pi[j];
        else i++;
    }

    delete[] pi;
    return -1;
}
```