Strings

Bjarki Ágúst Guðmundsson
Tómas Ken Magnússon
Árangursrík forritun og lausn verkefna

School of Computer Science
Reykjavík University
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
Example:

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

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$
- Three occurrences:
Example:

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

- Three occurrences:
  - $\text{cabcababacaba}$
Example:

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

- Three occurrences:
  - $\text{cabcababacaba}$
  - $\text{cabcababacaba}$
  - $\text{cabcababacaba}$
Example:

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

• Three occurrences:
  • cabcabcabacaba
  • 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

- $S$: bacbababaabcabab
- $T$: ababaca
Naive string matching algorithm

- $S$: bacbababaabcbab
- $T$: ababaca
Naive string matching algorithm

- $S$: bacbababaabcbab
- $T$: ababaca
Naive string matching algorithm

- $S$: bacbababaabcocabab
- $T$: ababaca
Naive string matching algorithm

```cpp
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?
The KMP algorithm avoids useless comparisons:

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

- \( S: \text{bacbababaabcbab} \)
- \( T: \text{ababaca} \)
Knuth–Morris–Pratt algorithm

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

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

- $S$: bacbabaabcbab
- $T$: ababaca
The KMP algorithm avoids useless comparisons:

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>$\pi[i]$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>$\pi[i]$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

• 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:

- $S$: bacbabaabcbab
- $T$: ababaca

5 characters match, so $q = 5$

$\pi[q] = \pi[5] = 3$

Then shift $q - \pi[q] = 5 - 3 = 2$ positions

$S$: bacbababaabcbab

$T$: ababaca
Knuth–Morris–Pratt algorithm

- Example:
  - $S$: bacbabababcababa
  - $T$: 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
  - $S$: bacbababaabcbab
  - $T$: ababaca
Knuth–Morris–Pratt algorithm

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

```c
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;
}
```
Sets of strings

- We often have sets (or maps) of strings
- Insertions and lookups usually guarantee $O(\log n)$ comparisons
- But string comparisons are actually pretty expensive...
- There are other data structures, like tries, which do this in a more clever way
Tries
Tries
struct node {
    node* children[26];
    bool is_end;

    node() {
        memset(children, 0, sizeof(children));
        is_end = false;
    }
};
```c
void insert(node* nd, char *s) {
    if (*s) {
        if (!nd->children[*s - 'a'])
            nd->children[*s - 'a'] = new node();

        insert(nd->children[*s - 'a'], s + 1);
    } else {
        nd->is_end = true;
    }
}
```
bool contains(node* nd, char *s) {
    if (*s) {
        if (!nd->children[*s - 'a'])
            return false;
        return contains(nd->children[*s - 'a'], s + 1);
    } else {
        return nd->is_end;
    }
}
node *trie = new node();

insert(trie, "banani");

if (contains(trie, "banani")) {
    // ...
}
• Time complexity?

• Let $k$ be the length of the string we’re inserting/looking for

• Lookup and insertion are both $O(k)$
• Say we’re dealing with some string $S$ of length $n$

• Let’s insert all suffixes of $S$ into a trie

• $S = \text{banani}$
  
  • insert(trie, "banani");
  • insert(trie, "anani");
  • insert(trie, "nani");
  • insert(trie, "ani");
  • insert(trie, "ni");
  • insert(trie, "i");
There are a lot of cool things we can do with suffix tries

Example: String matching

If a string $T$ is a substring in $S$, then (obviously) it has to start at some suffix of $S$

So we can simply look for $T$ in the suffix trie of $S$, ignoring whether the last node is an end node or not

This is just $O(m)$...
String matching is fast if we have the suffix trie for $S$.

But what is the time complexity of suffix trie construction?

There are $n$ suffixes, and it takes $O(n)$ to insert each of them.

So $O(n^2)$, which is pretty slow.

Can we do better?

There can be up to $n^2$ nodes in the graph, so this is actually optimal...
- There exists a compressed version of a suffix trie, called a suffix tree.
- It can be constructed in $O(n)$, and has all the features that suffix tries have.
- But the $O(n)$ construction algorithm is pretty complex, a big disadvantage for us.
• A variation of the previous structures
• Can do everything the other structures can do, with a small overhead
• Can be constructed pretty quickly with relatively simple code
Suffix arrays

- Take all the suffixes of $S$

  banani
  anani
  nani
  ani
  ni
  i

- and sort them

  anani
  ani
  banani
  i
  nani
  ni
Suffix arrays

- We can use this array to do everything that suffix tries can do
- Like string matching
• Let's look for nan

anani
ani
banani
i
nani
ni
• Let's look for nan
• The first letter in the string has to be n, so we can binary search for the range of strings starting with n

anani
ani
banani
i
nani
ni
• Let’s look for nan
• The first letter in the string has to be $n$, so we can binary search for the range of strings starting with $n$

nani
ni
Suffix arrays

- Let’s look for nan
- The second letter in the string has to be a, so we can binary search for the range of strings that have a as the second letter

nani
ni
• Let’s look for nan
• The second letter in the string has to be a, so we can binary search for the range of strings that have a as the second letter

nani
Let’s look for nan

The third letter in the string has to be n, so we can binary search for the range of strings that have n as the third letter

nani
• Let’s look for nan
• The third letter in the string has to be n, so we can binary search for the range of strings that have n as the third letter

nani
• Let’s look for nan

• The third letter in the string has to be n, so we can binary search for the range of strings that have n as the third letter

nani

• If there is at least one string left, we have a match
Suffix arrays

- Time complexity?
- For each letter in $T$, we do two binary searches on the $n$ suffixes to find the new range
- Time complexity is $O(m \times \log n)$
- A bit slower than doing it with a suffix trie, but still not bad
• But how do we construct a suffix array for a string?

• A simple `sort(suffixes)` is $O(n^2 \log(n))$, because comparing two suffixes is $O(n)$

• And we still have the same problem as with suffix tries, there are almost $n^2$ characters if we store all suffixes
The second problem is easy to fix
Just store the indices of the suffixes

- anani
- ani
- banani
- i
- nani
- ni

- becomes

1: anani
3: ani
0: banani
5: i
2: nani
4: ni
Suffix arrays

• What about the construction?
• In short, we
  • sort all suffixes by only looking at the first letter
  • sort all suffixes by only looking at the first 2 letters
  • sort all suffixes by only looking at the first 4 letters
  • sort all suffixes by only looking at the first 8 letters
  • ...
  • sort all suffixes by only looking at the first \(2^i\) letters
  • ...

• If we use an \(O(n \log n)\) sorting algorithm, this is \(O(n \log^2 n)\)
• We can also use an \(O(n)\) sorting algorithm, since all sorted values are between 0 and \(n\), bringing it down to \(O(n \log n)\)
struct suffix_array {
    struct entry {
        pair<int, int> nr;
        int p;

        bool operator <(const entry &other) const {
            return nr < other.nr;
        }
    };

    string s;
    int n;
    vector<vector<int> > P;
    vector<entry> L;
    vi idx;

    // constructor
};
suffix_array(string _s) : s(_s), n(s.size()) {
    L = vector<entry>(n);
    P.push_back(vi(n));
    idx = vi(n);

    for (int i = 0; i < n; i++) {
        P[0][i] = s[i];
    }

    for (int stp = 1, cnt = 1; (cnt >> 1) < n; stp++, cnt <<= 1) {
        P.push_back(vi(n));
        for (int i = 0; i < n; i++) {
            L[i].p = i;
            L[i].nr = make_pair(P[stp - 1][i], i + cnt < n ? P[stp - 1][i + cnt] : -1);
        }
        sort(L.begin(), L.end());
        for (int i = 0; i < n; i++) {
            if (i > 0 && L[i].nr == L[i - 1].nr) {
                P[stp][L[i].p] = P[stp][L[i - 1].p];
            } else {
                P[stp][L[i].p] = i;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        idx[P[P.size() - 1][i]] = i;
    }
}
There is also one other useful operation on suffix arrays:

- Finding the longest common prefix (lcp) of two suffixes of $S$

$S = \text{anani}$

- $lcp(1,3) = 2$
- $lcp(2,1) = 0$

This function can be implemented in $O(\log n)$ by using intermediate results from the suffix array construction.
int lcp(int x, int y) {
    int res = 0;
    if (x == y) return n - x;
    for (int k = P.size() - 1; k >= 0 && x < n && y < n; k--) {
        if (P[k][x] == P[k][y]) {
            x += 1 << k;
            y += 1 << k;
            res += 1 << k;
        }
    }
    return res;
}
Longest common substring

- Given two strings $S$ and $T$, find their longest common substring
  - $S = \text{banani}$
  - $T = \text{kanina}$
  - Their longest common substring is $\text{ani}$

- see example