Strings

Zachary Friggstad

Programming Club Meeting

Outline

- Suffix Arrays
- Knuth-Morris-Pratt Pattern Matching

Suffix Arrays (no code, see Comp. Prog. text)

Sort all of the *suffixes* of a string lexicographically.

`bananaban`

- `aban`
- `an`
- `anaban`
- `ananaban`
- `ban`
- `bananaban`
- `n`
- `naban`
- `nanaban`

Obvious how to do it: $O(n^2 \log n)$ - create all suffixes and sort them.

Obvious how to do it: $O(n^2 \log n)$ - create all suffixes and sort them.

Can actually get $O(n)$ time!

Obvious how to do it: $O(n^2 \log n)$ - create all suffixes and sort them.

Can actually get $O(n)$ time!

This is often overkill in the contest setting and a bit technical, let's see an $O(n \log^2 n)$ algorithm.

But writing all suffixes takes $\Theta(n^2)$ space, we need a **compact** representation.

Obvious how to do it: $O(n^2 \log n)$ - create all suffixes and sort them.

Can actually get $O(n)$ time!

This is often overkill in the contest setting and a bit technical, let's see an $O(n \log^2 n)$ algorithm.

But writing all suffixes takes $\Theta(n^2)$ space, we need a **compact** representation.

Suffix Array
An array of indices of the start positions of the suffixes in sorted order.

Example
For string `bananaban`
`int sarray[] = {5, 7, 3, 1, 6, 0, 8, 4, 2};`

Idea: For $i = 0, \ldots, \log_2 n$, sort the suffixes just by their first 2^i characters.

$i = 0$

- `ananaban`
- `anaban`
- `aban`
- `an`
- `bananaban`
- `ban`
- `nanaban`
- `naban`
- `n`

Can do in $O(n \log n)$ time (recall we are actually just sorting the indices, not the whole suffixes).

Next, sort the suffixes by their length 2 prefixes.

- `aban`
- `ananaban`
- `anaban`
- `an`

- `bananaban`
- `ban`

- `n`
- `nanaban`
- `naban`

Next, sort the suffixes by their length 4 prefixes.

- aban
- an
- anaban
- ananaban
- ban
- bananaban
- n
- naban
- nanaban

To check if nanaban $<$ naban, just look up the 2nd half of the red parts to see how they were ordered last step.

Generally, to sort the suffixes by their length 2^{i+1} prefixes we check $<$ using the ordering based on length 2^i prefixes.

Generally, to sort the suffixes by their length 2^{i+1} prefixes we check $<$ using the ordering based on length 2^i prefixes.

Check how the first 2^i characters of two suffixes a, b compare using the previous ordering. If they are different then just return that result.

If they are the same, check how the second 2^i characters of a, b compare again using the previous ordering.

Generally, to sort the suffixes by their length 2^{i+1} prefixes we check $<$ using the ordering based on length 2^i prefixes.

Check how the first 2^i characters of two suffixes a, b compare using the previous ordering. If they are different then just return that result.

If they are the same, check how the second 2^i characters of a, b compare again using the previous ordering.

Example

`nanaban` vs. `naban`. Length-2 prefixes are the same (`na`), but next 2 characters (`na` vs. `ba`) show the answer is $>$.

Sorting based on length 2^i prefixes then takes only $O(n \log n)$ time. Since i ranges up to $\log_2 n$, then overall time is $O(n \log^2 n)$.

Can also quickly compute the longest common prefix between adjacent suffixes in the array.

Can also quickly compute the longest common prefix between adjacent suffixes in the array.

Example
`naban` and `nanaban` are adjacent suffixes in the suffix array.

Their common prefix length is 2. This information can easily be construct along with the construction of the suffix array itself.

Can also quickly compute the longest common prefix between adjacent suffixes in the array.

Example
`naban` and `nanaban` are adjacent suffixes in the suffix array.

Their common prefix length is 2. This information can easily be construct along with the construction of the suffix array itself.

Faster Algorithm
Getting down to $O(n)$ running time is a bit of a pain, but $O(n \log n)$ isn't so bad.

We can "bucket sort" each step in $O(n)$ time if we have an appropriate mapping of the length 2^{i-1} substrings to integers $\{0, \ldots, n-1\}$.

Knuth-Morris-Pratt

Given a source string s and a pattern string p, does p appear as a substring of a?

Knuth-Morris-Pratt

Given a source string s and a pattern string p, does p appear as a substring of a?

More generally, record all positions i such that p appears as a substring of a starting at position i.

Example

`s = findmatchingmatches`

`p = match`

Then p appears as a substring of s at indices 4 and 12 (highlighted).

An obvious algorithm is to try all locations of s and linearly scan to see if p matches there.

Can take $\Omega(|s| \cdot |p|)$ time. The Knuth-Morris-Pratt (KMP) algorithm only takes $O(|s| + |p|)$ time!

An obvious algorithm is to try all locations of s and linearly scan to see if p matches there.

Can take $\Omega(|s| \cdot |p|)$ time. The Knuth-Morris-Pratt (KMP) algorithm only takes $O(|s| + |p|)$ time!

Main Idea: For each index i into p, let $\pi[i]$ denote the length of the longest proper suffix of $p_0 p_1 \ldots p_i$ that is also a prefix of p.

An obvious algorithm is to try all locations of s and linearly scan to see if p matches there.

Can take $\Omega(|s| \cdot |p|)$ time. The Knuth-Morris-Pratt (KMP) algorithm only takes $O(|s| + |p|)$ time!

Main Idea: For each index i into p, let $\pi[i]$ denote the length of the longest proper suffix of $p_0 p_1 \ldots p_i$ that is also a prefix of p.

Confusing? Example!

`p = acabaca`

The longest proper suffix of `acabac` that is also a prefix is `ac`.

```
pi[] = {0, 0, 1, 0, 1, 2, 3};
```

Slide the pattern p "over" s.

```
acabaca
acacabacabaca
```

Slide the pattern p “over” s.

`acabaca`
`acacabacabaca`

Match as many symbols as possible

`acabaca`
`acacabacabaca`

Slide the pattern p “over” s.

```
acabaca
acacabacabaca
```

Match as many symbols as possible

```
acabaca
acacabacabaca
```

When stuck, slide the pattern to the next partial match.

```
  acabaca
acacabacabaca
```

Distance to slide encoded by prefix table π.

Continue matching

```
  acabaca
acacabacabaca
```

Continue matching

```
  acabaca
acacabacabaca
```

Found a match, record it! Slide pattern over to the next partial match.

```
      acabaca
acacabacabaca
```

Continue matching

```
  acabaca
acacabacabaca
```

Found a match, record it! Slide pattern over to the next partial match.

```
      acabaca
acacabacabaca
```

Continue matching

```
      acabaca
acacabacabaca
```

Another match, record it!

Slide pattern over to next partial match.

```
          acabaca
acacabacabaca
```

Slide pattern over to next partial match.

```
          acabaca
acacabacabaca
```

Quit, the pattern is past the end of the string.

Slide pattern over to next partial match.

```
          acabaca
acacabacabaca
```

Quit, the pattern is past the end of the string.

Runs in $O(|s| + |p|)$ time because each step increases the "matched" pointer or slides the pattern over. Each slide takes $O(1)$ time using the π values.

```
void kmp(const string& s, const string& p) {
    vector<int> pi;
    compute_prefix(p, pi); //next two slides :)

    // invariant: at the start of each iteration hit is the
    // length of the longest *proper* prefix of p[] that
    // matches the suffix of s[0...(i-1)]
    for (int i = 0, hit = 0; i < s.length(); ++i) {
        // slide the window until a hit (or slid past)
        while (hit > -1 && p[hit] != s[i]) hit = pi[hi];

        // or do whatever to process the match, just
        // make sure hit is incremented for sure and is
        // shifted back to p[hit] if there is a match
        if (++hit == p.length()) {
            cout << "Match:" << i << endl;
            hit = pi[hit];
        }
    }
}
```

How to compute π? Basically the same idea!

How to compute π? Basically the same idea!

`p = bananaban`

Note, a suffix of $\pi[i]$ that is also a prefix of p comes from a suffix of $\pi[i-1]$ that is a prefix of p.

`bananaban`: Note `ba` is a suffix of `bananaba` that is also a prefix of p.

How to compute π? Basically the same idea!

`p = bananaban`

Note, a suffix of $\pi[i]$ that is also a prefix of p comes from a suffix of $\pi[i-1]$ that is a prefix of p.

`bananaban`: Note `ba` is a suffix of `bananaba` that is also a prefix of p.

So,

- $\pi[i]$ is just $\pi[i-1]$ if $s[i] == s[\pi[i-1]]$.
- Otherwise, check $s[i] == s[\pi[\pi[i-1]]]$ and so on.

How to compute π? Basically the same idea!

`p = bananaban`

Note, a suffix of $\pi[i]$ that is also a prefix of p comes from a suffix of $\pi[i-1]$ that is a prefix of p.

`bananaban`: Note `ba` is a suffix of `bananaba` that is also a prefix of p.

So,

- $\pi[i]$ is just $\pi[i-1]$ if $s[i] == s[\pi[i-1]]$.
- Otherwise, check $s[i] == s[\pi[\pi[i-1]]]$ and so on.

Overall idea: slide the pattern over itself!

```
    acabaca
acabaca
```

```
void compute_prefix(const string& p, vector<int>& pi) {
  pi.resize(p.length()+1);
  pi[0] = -1;

  for (int i = 0; i < p.length(); ++i) {
    // start with the shift from the previous character
    pi[i+1] = pi[i];

    // slide the window until the next character matches
    while (pi[i+1] > -1 && p[pi[i+1]] != p[i])
      pi[i+1] = pi[pi[i+1]];

    // we matched a character or slid back to index -1
    // in either case, increment
    ++pi[i+1];
  }
}
```

Missing Topics

- Tries (presented later as a CMPUT 403 project topic)
- Suffix Trees
- Manachar's Algorithm: find all *maximal palindromes* in linear time.

Next Week

Bipartite graphs: recognition, matching, and edge colouring.

Open Kattis - lifeforms

Open Kattis - lifeforms

Starting Question: How can you find the longest substring in common with 2 strings s, t?

Open Kattis - lifeforms

Starting Question: How can you find the longest substring in common with 2 strings s, t?

Concatenate $s \cdot t$ and form a suffix array. Find the largest $LCP[i]$ value where $i, i+1$ come from different strings s, t.

continued next slide

Open Kattis - lifeforms

For more than two strings $s^1, s^2, \ldots, s^k$, form the suffix array for $s^1 \cdot s^2 \cdot \ldots \cdot s^k$. What can we do?

Open Kattis - lifeforms

For more than two strings $s^1, s^2, \ldots, s^k$, form the suffix array for $s^1 \cdot s^2 \cdot \ldots \cdot s^k$. What can we do?

For indices i, j into the suffix array, we can compute the number of *distinct* strings s^i represented by these indices.

Open Kattis - lifeforms

For more than two strings $s^1, s^2, \ldots, s^k$, form the suffix array for $s^1 \cdot s^2 \cdot \ldots \cdot s^k$. What can we do?

For indices i, j into the suffix array, we can compute the number of *distinct* strings s^i represented by these indices.

For every $i \leq$ representing $> k/2$ lifeforms, compute $\min_{i \leq \ell < k} LCP[\ell]$.

Open Kattis - lifeforms

For more than two strings $s^1, s^2, \ldots, s^k$, form the suffix array for $s^1 \cdot s^2 \cdot \ldots \cdot s^k$. What can we do?

For indices i, j into the suffix array, we can compute the number of *distinct* strings s^i represented by these indices.

For every $i \leq$ representing $> k/2$ lifeforms, compute $\min_{i \leq \ell < k} LCP[\ell]$.

Examine "minimal" pairs i, j (no "smaller" pair for $> k/2$ lifeforms).

Open Kattis - lifeforms

For more than two strings $s^1, s^2, \ldots, s^k$, form the suffix array for $s^1 \cdot s^2 \cdot \ldots \cdot s^k$. What can we do?

For indices i, j into the suffix array, we can compute the number of *distinct* strings s^i represented by these indices.

For every $i \leq$ representing $> k/2$ lifeforms, compute $\min_{i \leq \ell < k} LCP[\ell]$.

Examine "minimal" pairs i, j (no "smaller" pair for $> k/2$ lifeforms).

Use a **sliding window**: Increment j each "outer" iteration. For each j, while $(i+1, j)$ represents $> k/2$, then increment i.

Open Kattis - lifeforms

For more than two strings $s^1, s^2, \ldots, s^k$, form the suffix array for $s^1 \cdot s^2 \cdot \ldots \cdot s^k$. What can we do?

For indices i, j into the suffix array, we can compute the number of *distinct* strings s^i represented by these indices.

For every $i \leq$ representing $> k/2$ lifeforms, compute $\min_{i \leq \ell < k} LCP[\ell]$.

Examine "minimal" pairs i, j (no "smaller" pair for $> k/2$ lifeforms).

Use a **sliding window**: Increment j each "outer" iteration. For each j, while $(i+1, j)$ represents $> k/2$, then increment i.

Use a heap to hold the $LCP[\ell]$ values for $i \leq \ell < j$. Pop the min if it is irrelevant (i.e. $\ell < i$).

Open Kattis - bugs

Open Kattis - bugs

Can identify all occurrences of the bug in a line s in $O(|s|)$ (after building the prefix table for the bug).

Open Kattis - bugs

Can identify all occurrences of the bug in a line s in $O(|s|)$ (after building the prefix table for the bug).

But what about when a bug is removed? It may introduce a new bug!

Open Kattis - bugs

Can identify all occurrences of the bug in a line s in $O(|s|)$ (after building the prefix table for the bug).

But what about when a bug is removed? It may introduce a new bug!

Can we somehow "resume" the KMP process after removing a bug?

Open Kattis - bugs

Can identify all occurrences of the bug in a line s in $O(|s|)$ (after building the prefix table for the bug).

But what about when a bug is removed? It may introduce a new bug!

Can we somehow "resume" the KMP process after removing a bug?

One a bug is removed, rematch the longest possible prefix from the previous unremoved character to resume KMP.

Open Kattis - bugs

Can identify all occurrences of the bug in a line s in $O(|s|)$ (after building the prefix table for the bug).

But what about when a bug is removed? It may introduce a new bug!

Can we somehow "resume" the KMP process after removing a bug?

One a bug is removed, rematch the longest possible prefix from the previous unremoved character to resume KMP.

Can do in $O(1)$ time if we just remember the longest match at each character.

Open Kattis - bugs

Can identify all occurrences of the bug in a line s in $O(|s|)$ (after building the prefix table for the bug).

But what about when a bug is removed? It may introduce a new bug!

Can we somehow "resume" the KMP process after removing a bug?

One a bug is removed, rematch the longest possible prefix from the previous unremoved character to resume KMP.

Can do in $O(1)$ time if we just remember the longest match at each character.

Should also keep track of the next and previous unremoved character for each letter to "jump" the gaps in $O(1)$ time while scanning.

Open Kattis - chasingsubs

Open Kattis - chasingsubs

Idea: use KMP but “construct” the permutation on the fly.

Open Kattis - chasingsubs

Idea: use KMP but "construct" the permutation on the fly.

Still build π for the pattern, but also `prev[]` mapping an index i to the previous occurrence of the same letter in the pattern.

Open Kattis - chasingsubs

Idea: use KMP but "construct" the permutation on the fly.

Still build π for the pattern, but also `prev[]` mapping an index i to the previous occurrence of the same letter in the pattern.

In the KMP matching stage (i.e. sliding the pattern over the text), also keep track of the permutation of the letter so far.

Open Kattis - chasingsubs

Idea: use KMP but "construct" the permutation on the fly.

Still build π for the pattern, but also `prev[]` mapping an index i to the previous occurrence of the same letter in the pattern.

In the KMP matching stage (i.e. sliding the pattern over the text), also keep track of the permutation of the letter so far.

When we trying to match $p[i]$ to $s[j]$ when sliding the pattern, if $prev[i]$ is defined then matched ensure $p[prev[i]] = s[j]$. Otherwise, define the encryption permutation to send $p[i]$ to $s[j]$.

Open Kattis - chasingsubs

Idea: use KMP but "construct" the permutation on the fly.

Still build π for the pattern, but also `prev[]` mapping an index i to the previous occurrence of the same letter in the pattern.

In the KMP matching stage (i.e. sliding the pattern over the text), also keep track of the permutation of the letter so far.

When we trying to match $p[i]$ to $s[j]$ when sliding the pattern, if $prev[i]$ is defined then matched ensure $p[prev[i]] = s[j]$. Otherwise, define the encryption permutation to send $p[i]$ to $s[j]$.

When sliding the pattern because of "no match", remove rules from the encryption permutation as you slide past characters. There are some details to consider here, but it can be done in linear time.