Suffix Trees and Arrays

Yufei Tao

KAIST

May 8, 2013

We will discuss the following substring matching problem:

Problem (Substring Matching)

Let σ be a single string of n characters. Given a query string q, a query returns:

- all the starting positions of q in s, if q is a substring of σ;
- nothing otherwise.

Example

Let $\sigma = \mathtt{aabbabab}$. Then:

- for $q = \mathtt{abb}$, return 2 because substring abb starts at the 2nd position of σ.
- for $q = \mathtt{bab}$, return 4 and 6.
- for $q = \mathtt{bbb}$, return nothing.

Relation to Search Engines

Our technique for solving this problem will give an elegant solution to the phrase matching problem, which is crucial to search engines:

Problem (Phrase Matching)

Let S be a set of documents, each of which is a sequence of characters, and has a distinct id. Given a sequence q of characters, a query returns the ids of all documents in S that contain q as a substring.

See the next slide for an example.

Relation to Search Engines (cont.)

Example

Let S include the following two documents:

- document 1: "`Search engines are not very effective for irregular queries.`"
- document 2: "`Without search engines, the Internet would not have been so popular.`"

Then:

- for $q =$ "`search engine`", return 1 and 2.
- for $q =$ "`very effective`", return 1.
- for $q =$ "`ular`", return 1 and 2.

Now, we return to the substring matching problem.

We will denote a string s as a sequence $s[1]s[2]...s[l]$ where $l = |S|$, and $s[i]$ is the i-th character of s $(1 \leq i \leq f)$.

Definition (Suffix)

For a string $s = s[1]s[2]...s[l]$, the string $s[i]s[i+1]...s[l]$ is called a suffix of s for each $i \in [1, l]$.

Clearly, a string s has $|s|$ suffixes.

Example

String aabbabab has 8 suffixes: aabbabab, abbabab, bbabab, babab, abab, bab, ab, b.

Recall that σ is the input string of our substring matching problem. We consider that σ is stored in an array of length $|\sigma|$. We will denote by S the set of suffixes of σ.

Lemma

A query string q is a substring of σ if and only if q is a prefix of a string in S.

Earlier, we proved:

Lemma

Let S be a set of m strings, each of which has a unique integer id. We can build a structure of $O(m)$ space such that, given a query string q, the ids of all strings $s \in S$ such that q is a prefix of s can be reported in $O(\log m + |q| + k)$ time, where k is the number of ids reported.

We thus immediately obtain:

Lemma

For the substring matching problem, we can build a structure of $O(n)$ space such that, given a query string q, the starting positions of all substring q in σ can be reported in $O(\log n + |q| + k)$ time, where k is the number of reported positions.

The structure implied by the above lemma is called the suffix tree – essentially, a patricia trie on all the suffixes of σ.

Remark

The lemma of the previous slide assumes that (i) every string in S is stored in an array, and (ii) every string terminates with a special symbol $\perp$. Convince yourself that neither assumption prevents us from deriving the above lemma.

The subsequent slides will not be tested in the quizzes and exams.

The suffix tree is not so easy to implement. Next, we will discuss an alternative structure called the suffix array that is more amenable to practical use.

To simplify discussion, let us focus on a slightly different problem:

Problem (Substring Detection)

Let σ be a single string of n characters. Given a query string q, a query returns:

- yes, if q is a substring of σ;
- no, otherwise.

The suffix array can be easily extended to solve the substring matching problem, which is left as an exercise for you.

As before, σ is the data input to the substring matching problem, and S is the set of all suffixes of σ. For each suffix $s \in S$, define its index to be the starting position of s in σ.

Example

Let $\sigma = \texttt{aabbabab}$. Then, the index of babab is 4, and that of bab is 6.

For any s, using its index, we can access any $s[i]$ for any $i \in [1, |s|]$ in constant time. Think: why? (Hint: use the array of σ)

Now, we sort S alphabetically in ascending order. From now on, we denote by $S[i]$, $1 \leq i \leq n$, the i-th string of S in the ordering.

We can represent the ordering using an array A of size n, specifically, by storing in $A[i]$ the index of $S[i]$.

See the next slide for an example.

Example

Let $\sigma = \texttt{aabbabab}$. Then:

- $S[1] = \texttt{aabbabab}$, $A[1] = 1$.
- $S[2] = \texttt{ab}$, $A[2] = 7$.
- $S[3] = \texttt{abab}$, $A[3] = 5$.
- $S[4] = \texttt{abbabab}$, $A[4] = 2$.
- $S[5] = \texttt{b}$, $A[5] = 8$.
- $S[6] = \texttt{bab}$, $A[6] = 6$.
- $S[7] = \texttt{babab}$, $A[7] = 4$.
- $S[8] = \texttt{bbabab}$, $A[8] = 3$.

Given a string q, we can answer a substring detection query by performing binary search on array S. For simplicity, we consider only the case where q is a prefix of neither $S[1]$ and $S[n]$ (think: otherwise, what do we do?). Here is the algorithm:

1. Suppose that we know q is greater a left string $S[l]$ but smaller than a right string $S[r]$ (at the beginning, $l = 1$ and $r = n$).
2. Call $S[m]$ the middle string, where $m = \lfloor (l + r)/2 \rfloor$.
3. If q is a prefix of $S[m]$, return yes.
4. Else if $q < S[m]$, then we only need to search in $S[l + 1], ..., S[m - 1]$. Hence, we set $r = m$ and go back to 2 if $l \leq r - 2$ (otherwise, return no).
5. Else, we only need to search in $S[m + 1], ..., S[r - 1]$. Set $l = m$ and go back to 2 if $l \leq r - 2$ (otherwise, return no).

The query time is $O(|q| \log n)$, because it takes $O(|q|)$ time to compare q to $S[m]$, and we need to do $O(\log n)$ comparisons.

Next, we will show how to improve the time of this algorithm substantially to $O(\log n + |q|)$.

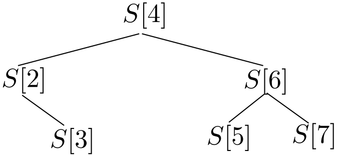
At each iteration of the binary search, we denote $[S[l], S[r]]$ as a search interval.

Crucial observation: there can be only n possible search intervals.

To show this, observe that binary search can be described instead by a decision tree T defined as follows:

- If $|S| \le 2$, then T is empty.
- Otherwise, the root u of T is $S[m]$ where $m = \lfloor (1 + n)/2 \rfloor$. Also, u has a left subtree T_1 and a right subtree T_2, where
  - T_1 is the decision tree on array $S[1, ..., m]$
  - T_2 is the decision tree on $S[m, ..., n]$.

The decision tree T when S has 8 elements:



In general, T has $n-2$ nodes. Furthermore, node $S[i]$ corresponds to a unique search interval $[S[l], S[r]]$ such that $i = \lfloor (l+r)/2 \rfloor$, that is, $S[i]$ is the middle string of this search interval.

To achieve query time $O(\log n + |q|)$, besides A, we need two more integer arrays L and R of size n.

For each $i \in [2, n-1]$, let $[left, right]$ be the search interval that $S[i]$ corresponds to. Then:

- $L[i]$ = the length of the longest common prefix (LCP) of *left* and $S[i]$.
- $R[i]$ = the LCP length of *right* and $S[i]$.

Example

Let $\sigma = \text{aabbabab}$. Then:

- $S[1] = \text{aabbabab}$, $A[1] = 1$.
- $S[2] = \text{ab}$, $A[2] = 7$, $L[2] = 1$, $R[2] = 2$.
- $S[3] = \text{abab}$, $A[3] = 5$, $L[3] = 2$, $R[3] = 2$.
- $S[4] = \text{abbabab}$, $A[4] = 2$, $L[4] = 1$, $R[4] = 0$.
- $S[5] = \text{b}$, $A[5] = 8$, $L[5] = 0$, $R[5] = 1$.
- $S[6] = \text{bab}$, $A[6] = 6$, $L[6] = 0$, $R[6] = 1$.
- $S[7] = \text{babab}$, $A[7] = 4$, $L[7] = 3$, $R[7] = 1$.
- $S[8] = \text{bbabab}$, $A[8] = 3$.

The suffix array on S is nothing but the collection of three arrays: A, L, and R. The total space consumption is clearly $O(n)$.

We now discuss how to answer a substring detection query with string q using a suffix array. The algorithm is the same as the one in Slide 14, except that at any point, we maintain two values:

- $lcplen(left, q)$:
  the LCP length of q and the left string $left = S[l]$.
- $lcplen(right, q)$:
  the LCP length of q and the right string $right = S[r]$.

Let us also define:

- $lcplen(left, mid)$:
  the LCP length of $left$ and the middle string $mid = s[m]$.
- $lcplen(right, mid)$:
  the LCP length of mid and $right$.

Knowing m, both $lcplen(left, mid)$ and $lcplen(right, mid)$ can be obtained as $L[m]$ and $R[m]$ in constant time, respectively.

Next, we show how to incrementally maintain $lcplen(left, q)$ and $lcplen(right, q)$.

Case 1: $lcplen(left, q) < lcplen(left, mid)$

It must hold that $q > mid$ (think: why?).

Hence, $l = m$; $lcplen(left, q)$ and $lcplen(right, q)$ remain unchanged (think: why?).

Case 2: $lcplen(left, q) > lcplen(left, mid)$

It must hold that $q < mid$ (think: why?). Thus, $r = m$, and we set:

$$lcplen(right, q) = lcplen(left, mid)$$

Think: Why is the above correct? Also, $lcplen(right, q)$ cannot have decreased; why?

The next two cases are symmetric to the previous two:

Case 3: $lcplen(right, q) < lcplen(right, mid)$
Case 4: $lcplen(right, q) > lcplen(right, mid)$

Their handling is similar to the handling of Cases 1 and 2. The details are left to you.

Case 5: $lcplen(left, q) = lcplen(left, mid)$ and $lcplen(left, q) = lcplen(right, mid)$

We know that q and mid share the same first

$$x = \max\{lcplen(left, q), lcplen(right, q)\}$$

characters. Nevertheless, we do not know yet which of them is greater.

To find out, we decide the LCP length y of mid and q. This requires only $y - x + 1$ comparisons, namely, comparing $left[i]$ and $q[i]$ for each $i \in [x + 1, y + 1]$.

We then have three more cases.

Case 5.1: $y = |q|$

So, q is a prefix of *mid*. The algorithm finishes and returns yes.

Case 5.2: $y < |q|$ and $q[y+1] < mid[y+1]$

So, $q < mid$. Thus, $r = m$. We set

$$lcplen(right, q) = y$$

Remark

$lcplen(right, q)$ cannot have decreased after the above because $y \geq x \geq lcplen(right, q)$. On the other hand, if $y > x$, $lcplen(right, q)$ has increased by at least $y - x$.

Case 5.3: $y < |q|$ and $q[y+1] > mid[y+1]$

So, $q > mid$. Thus, $l = m$. We set

$$lcplen(left, q) \quad = \quad y$$

Remark

$lcplen(left, q)$ cannot have decreased. If $y > x$, $lcplen(left, q)$ has increased by at least $y - x$.

Theorem

For the substring detection problem, we can create a suffix array on σ that occupies $O(n)$ space, and enables a query with string q to be answered in $O(\log n + |q|)$ time.