| | |
|---|---|
| **6.897: Advanced Data Structures** | Spring 2003 |

# Lecture 10 — March 31, 2003

*Prof. Erik Demaine* — *Scribe: Michael Walfish*

## 1 Overview

In this lecture we study string matching and related problems. There are several famous algorithms for string matching, and we briefly review some of them at the beginning of the lecture. Since this is a data structures course, we will focus on an approach that involves pre-processing the text to build a very useful data structure: the *suffix tree*.

After showing how to use suffix trees to solve string matching, we show how to use this data structure to solve a related problem: *document retrieval*. In order to solve document retrieval, we need to solve other algorithmic and data structures problems, including the *Range Minimum Query* (RMQ) problem and the *Least Common Ancestor* (LCA) problem.

The lecture concludes with the application of LCA to suffix trees to solve several problems related to string matching.

## 2 String Matching – Problem Definition

We can state this classic problem in algorithmic terms as follows: given a text $T$ and a pattern $P$, each of which is a string over an alphabet $\Sigma$, find *all* or *some* occurrences of $P$ in $T$ or declare that there are no matches.

Before we turn to suffix trees, a solution with a heavy data structures component, we briefly review some classic solutions to string matching.

## 3 String Matching – Classic Algorithmic Solutions

There are a number of existing solutions that range from the brute-force deterministic to the very elegant randomized.

### 3.1 Brute Force

This is the obvious approach: begin at the first position in $T$, and see whether $P$ matches by indexing through both strings. If $P$ doesn't match, then look at the second position of $T$, reset the index in $P$ and check whether $P$ matches by indexing through both strings again. The running time of this approach is $O(|T| \cdot |P|)$. If $|P|$ is not constant—if $|P|$ is the same order complexity as $|T|$, for example—then this approach could become quadratic.

Knowing that we cannot do better than linear time (since the "needle" could be anywhere in the "haystack"), we are motivated to search for linear solutions.

### 3.2 Knuth-Morris-Pratt [KMP77]

This algorithm runs in $O(|T|)$, which is the best one could hope for in order complexity terms. The essence of the algorithm is that $P$ becomes pre-processed into a finite automaton. The algorithm is complicated, and we won't cover it here, but students are encouraged to look it up if they have never seen it before.

### 3.3 Boyer-Moore [BM77]

This algorithm runs in $O(|T|)$ but sometimes runs in $O(|T|/|P|)$, if things work out. The idea is to keep a sliding window onto $T$ but to try matching from the back of $P$ to the front. The algorithm looks at letters that do match to figure out how much to advance the sliding window. If the algorithm is told that the back does not match, then in some cases, it may be possible to advance as many as $|P|$ spaces forward. An amortized analysis yields the running times just mentioned.

### 3.4 Rabin-Karp [KR87]

This is an elegant randomized algorithm that runs in $O(|T|)$ with high probability (usually abbreviated "*whp*" and defined below). The idea is to make a numerical fingerprint of $P$, where the fingerprint is taken modulo a prime, $p$. The algorithm maintains a sliding window on $T$ and compares the numeric fingerprint of the current window on $T$ to the pre-computed fingerprint of $P$. If these fingerprints match, the algorithm declares a match. Updating the numeric fingerprint when the window slides takes constant time, so the entire algorithm is $O(|T|)$. Failure occurs if the two fingerprints are equal modulo the prime but in fact represent different strings.

Why does it make sense to use a randomized algorithm if there are deterministic ones that provide the same order complexity with zero probability of failure? The reason is that the randomized one is often easier both to implement and reason about, and, if its running time can be stated to be "whp", then the chance of failure is negligible.

**Digress to define "with high probability":** An event $E$ is said to occur *with high probability (whp)* if:

$$\Pr\{E\} \geq 1 - \frac{1}{n^c}$$

for all $c \geq 1$, where $n$ is the problem size. In our case, $E$ is the event "the running time of Rabin-Karp is $O(|T|)$", $n$ is $|T| + |P|$, and $c$ should be able to take on any value without changing the order complexity (at worst, different values of $c$ should imply different constants absorbed by the $O(\cdot)$). In the case of Rabin-Karp, the constant $c$ determines the size of the prime modulus $p$, over which the algorithm takes fingerprints (since, the larger the prime, the lower the probability that two fingerprints are going to be confused modulo that prime). Different values of $c$ do not affect the running time or order complexity of the space required to run the algorithm[1]. In our case, $E$ failing means that the algorithm either returns a wrong answer or that it does not finish on time. The important thing about "whp" is that it can mean "as likely as you want", since the $c$ term can be whatever you want.

# 4 String Matching – Data Structures Approach

The above solutions all assumed that we could pre-process $P$ but not $T$. We will now alter that assumption and permit ourselves to pre-process $T$. Our approach relies on suffix trees, a powerful and flexible data structure on which we will spend a good portion of the lecture. Below, we describe what a suffix tree is, how it can be used for string matching, and its utility in several other problems, including document retrieval.

## 4.1 Suffix Trees – Description

Suffix trees have come up a number of times in the literature [Wei73, McC76, Ukk95, GK97]. A **suffix tree** is a kind of **trie**. Let's first review tries.

**Description 1.** *A* **trie** *is a tree in which each node has children labeled by letters in* $\Sigma$.

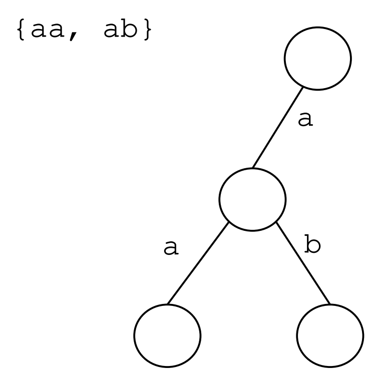


Figure 1: Trie storing strings $aa$ and $ab$

There is a very natural correspondence between tries and sets of strings over $\Sigma$, as shown in Figure 1. To avoid ambiguities that result from situations in which one of the members of the set is a prefix of another member of the set, we introduce a special symbol not in $\Sigma$: \$. So we would represent the set $\{a\$, aa\$\}$ with a trie as in Figure 2.

Note that every leaf is connected to the rest of the trie by the \$ symbol. Note, also, that this approach is wasteful: if a path does not branch, then we can store it as a single edge. This leads us to another description:

**Description 2.** *A* **compressed trie** *is a* **trie** *in which non-branching paths are stored as single edges and in which the edges are labeled with strings and not letters.*

[1]For a given value of $c$, the success probability is $\geq 1 - 1/n^c$ when we take $p = \Theta(n^{c+1} \log n^{c+1})$. (Also, $p$ must be $\leq 2^n$.) Note that for all values of $c$, $p$ can be represented with $\Theta(\log n + \log\log n)$ bits. For more detail, see Section 7.4 of Motwani and Raghavan's book *Randomized Algorithms*.

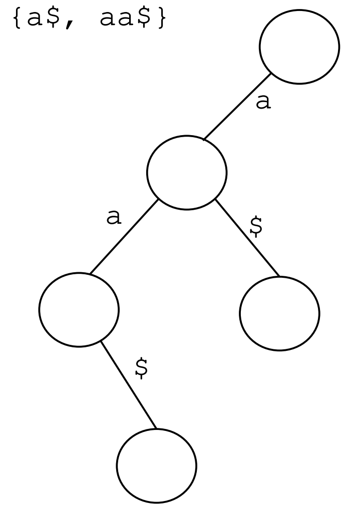


Figure 2: Trie storing $a$ and $aa$ with terminating symbol

See Figure 3 for an example of a compressed trie.

We are now ready to describe suffix trees, which will be our sledgehammer for attacking string matching and related problems. As we see below, a suffix tree is a very natural kind of trie.

**Description 3.** *A* **suffix tree** *is a compressed trie that stores a set of strings equal to all* $|T|+1$ *suffixes of the text string* $T$*.*

Note that we add 1 to $|T|$ because the empty string is a suffix. For an example of a suffix tree, see Figure 4. Before showing how suffix trees are useful for string matching and related problems, such as document retrieval, we discuss the **cost** of suffix trees. We would like it to be the case that constructing a suffix tree costs linear time and that storing the suffix tree costs linear space. The reason is that $|T|$ could be something extremely large, such as a document repository or massive database. In these cases, we cannot afford quadratic space or time.

**Suffix trees: Space cost** If each edge of the suffix tree is labeled with a string, then we can wind up storing a number of letters quadratic in $|T|$. As an example, imagine the suffix tree for the string $abcdefgh \ldots z$—in that case, we will have to store \$, $z\$$, $yz\$$, and every suffix up to $abc \ldots xyz\$$. To reduce our space consumption to something linear in $|T|$, we simply store at each edge the indices $i, j$, where $i, j$ correspond to the start and end indices of the substring that is supposed to be labeling the edge. As a quick proof that we will only consume no more than linear space, note that, by construction, every internal node branches, which means there are fewer internal nodes than leaf nodes. But there are no more leaf nodes than suffixes, and there are a linear number of suffixes. Since every edge gets a constant-sized label (two indices), we are done.

**Suffix trees: Linear cost** We can build suffix trees in $O(|T|)$ time. We will cover the procedure for building a suffix tree in the next lecture.

The point to remember is that if we are pre-processing $T$, then suffix trees are as cheap as you could want, in order complexity terms—they require linear time to construct, and they consume linear space.

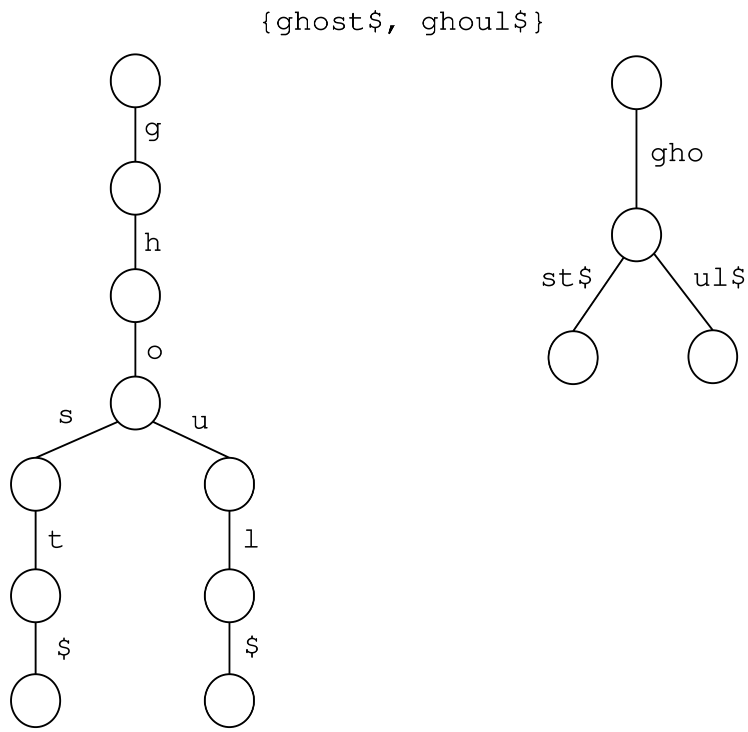


Figure 3: Uncompressed and compressed tries for *ghost* and *ghoul*

We will now show how to use suffix trees to solve the string matching problem.

## 4.2 Suffix Trees – Solving String Matching

**Claim 4.** *Given a pattern $P$, and a suffix tree $S$ for text $T$, we can find an occurrence of $P$ in time $O(|P|)$.*

*Proof.* We first outline our procedure and then discuss its correctness.

The idea is that we start at the root of $S$ and index through $P$, using the current letter of $P$ to decide which branch to take from the current node of $S$. When edges in $S$ are labeled with multiple letters, we index through the edge and through $P$, matching the edge labels to the current letter of $P$. We continue in this fashion, taking branches whenever we have indexed to the end of the current edge label. The algorithm stops when either: a) we have finished iterating through $P$, in which case we have a match, and we note the current node (that is, the node below the current edge) or b) the current position of $P$ does not match the current letter at the current edge, in which case there is no match or c) it is time to branch, but there is no branch that corresponds to the current letter of $P$, in which case there is no match. Figure 5 illustrates this procedure.

The correctness of this algorithm, namely that the algorithm declares a match if and only if $P$ occurs in $T$, is easily seen from the facts that a) $S$ stores exactly the set of all suffixes and b) any occurrence of $P$, say at position $i$, corresponds to a suffix, say $T[i{:}]$[2]. A formal proof is omitted.

[2]The notation $T[i{:}]$ denotes the substring of $T$ that starts at position $i$ and goes to the end of the string

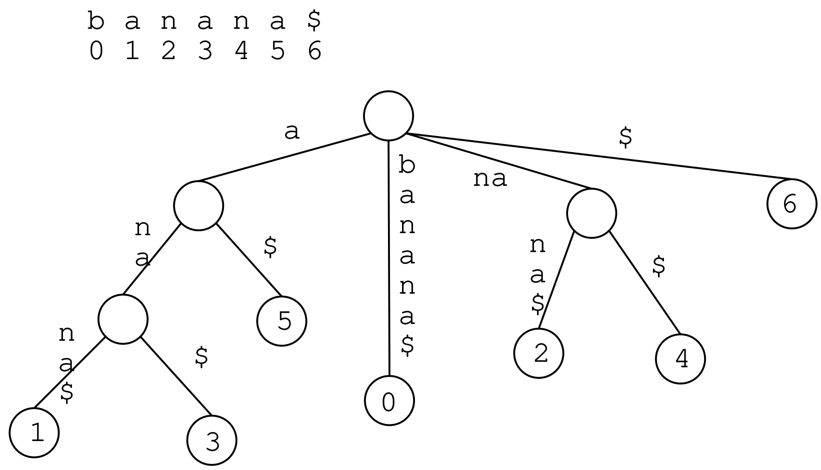


Figure 4: Suffix tree for *banana*

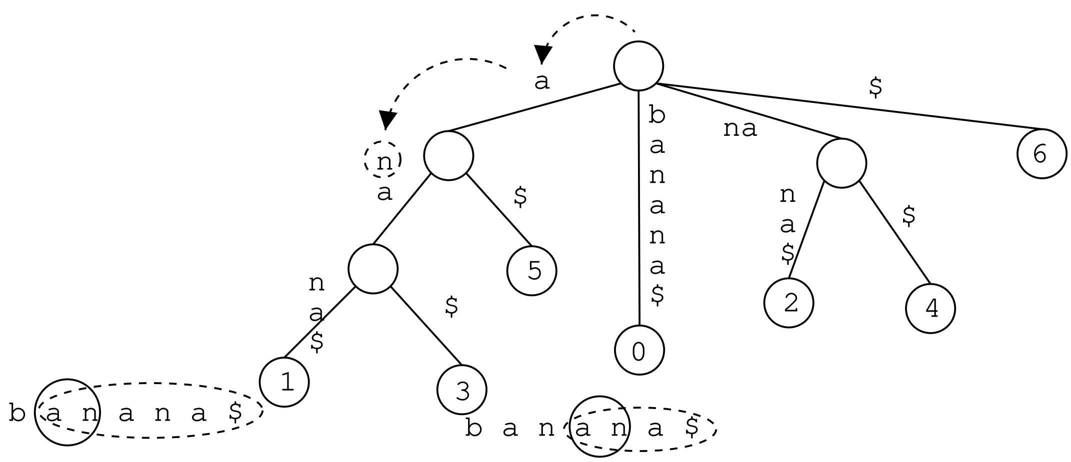


Figure 5: Lookup for *an* in suffix tree for *banana*

This algorithm clearly runs in time $O(|P|)$, since we are indexing through $P$, so we're done with the claim.

We note that the algorithm returns a subtree (since it returns a pointer to a node), and that the nodes of this returned subtree correspond to all of the occurrences of $P$ in $T$. To see this, imagine that $P$ occurs at indices $i_1, i_2, ...i_k$. Now consider the strings $T[i_1:]$, $T[i_2:]$, ... $T[i_k:]$. Each of these strings is a suffix of $T$ and so is stored in the suffix tree. Moreover, each of these suffixes begins with the same string (namely $P$), and so each of these suffixes will have the same ancestors in the tree (by the construction of the compressed trie). In other words, all of these substrings will be part of the same subtree. Given that the algorithm actually returns a subtree in answer to the question, "where does $P$ occur in $T$?", we can note the following properties of the procedure we have just outlined:

- We can extract $k$ matches in $O(k)$ time, so we do not need to do all of the work upfront to begin reporting matches. We simply iterate over the returned subtree and find matches as we go.

- Once we have looked at the pattern, we can extract the number of matches in $O(1)$ time (assuming we have augmented the tree by storing, at each node, how many children are below it).
- This data structure can answer a number of questions. One is: "what is the longest repeated substring that appears twice?". To answer this, we augment the tree by labeling each of the internal nodes with a "pattern depth"—the "pattern depth" is the number of letters on the path from the node to the root. Since all internal nodes have at least two children, by construction, then the internal node with the largest pattern depth corresponds to the longest substring that appears at least twice. It takes time $O(|T|)$ to find the non-leaf with the largest pattern depth label.

We note that this last question is conceptually several questions rolled into one and that we can use the structure of the suffix tree to answer these questions in parallel, in a sense. In the next section, we illustrate how to use suffix trees to solve a different, but related, problem: document retrieval.

# 5 Document Retrieval

This portion of the lecture is derived from a paper by Muthukrishan [Mut02].

**Motivation.** If we have a large collection of documents that we are permitted to pre-process, we might be interested in quickly answering queries about which documents contain a particular pattern. This is the problem that Web search engines face.

Formally, the problem can be stated as follows: given texts $T_1, T_2, \ldots, T_k$, we would like to pre-process the $T_i$ so that we can quickly answer queries of the form: in which of the $T_i$ does a pattern $P$ occur?

## 5.1 Using Suffix Trees for Document Retrieval

Motivated by the contents of Section 4.2, we might want to create a generalized suffix tree, using as our set of strings the suffixes of all $k$ text strings. In order to ensure that the trie does not confuse substrings from different texts, we assign a different terminator symbol to each text string—$\$_1, \$_2, \ldots, \$_k$. Using different termination symbols ensures that if $P$ occurs in different documents, we will be able to distinguish the different occurrences, and it preserves the property that all occurrences of $P$ will be rooted at the same subtree.

But this approach is not exactly what we want. The problem statement calls for identifying *which* documents contain $P$, not identifying *all occurrences* of $P$, which could potentially be much costlier (since it takes $O(m)$ time to return $m$ matches by using a suffix tree).

## 5.2 Reduction to Range Minimum Query

Ideally, what we would like, as illustrated in Figure 6, is a list of the different $\$_i$ appearing at the leaves of the particular subtree returned by the string matching algorithm (as described in the proof

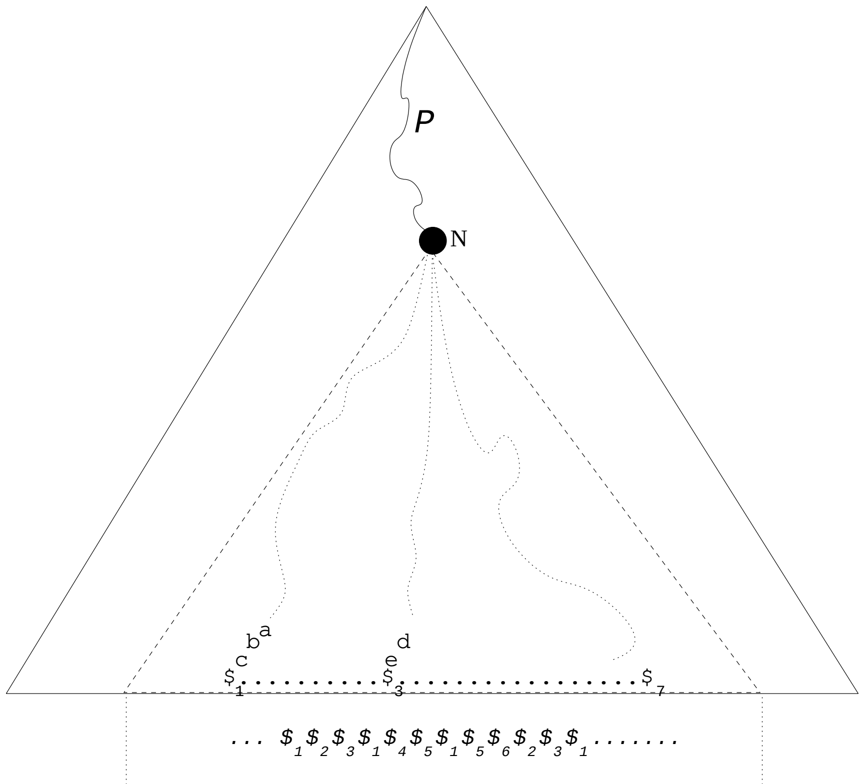


Figure 6: subtree rooted at N represents all matches of pattern $P$—for document retrieval, we need only a list of the distinct terminating symbols

of Claim 4). In the remainder of this section, we will reduce our problem—namely finding a list of distinct $\$_i$ in the set of leaves of a particular subtree—to the problem of finding the minimum number in a given interval, which we call the *Range Minimum Query* problem. The construction is as follows: use two parallel arrays $A, B$ that are each of size $|T|+1$. Each element in these arrays corresponds to a particular leaf (and since each leaf corresponds to a suffix, we need an array no bigger than $|T|+1$). We require that the elements that correspond to leaves of a subtree are stored consecutively. In the first of these arrays, $A$, we store, for each leaf, the index $d$ of the corresponding text $T_d$ whose suffix ends at this leaf (every leaf has a $\$_d$ symbol, so just grab the $d$.)

**The document retrieval problem can then be phrased as follows:** given the appropriate subtree (and its descendant leaves) of the suffix tree, and given the indices $i, j$ that demarcate the corresponding array elements in $A$, we would like to output a list of the distinct elements in the interval $A[i]$ to $A[j]$.

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | . . . |
|---|---|---|---|---|---|---|---|---|
| array A | 5 | 2 | 1 | 3 | 1 | 1 | 5 | . . |
| array B | -1 | -1 | -1 | -1 | 2 | 4 | 0 | . . |

Figure 7: We keep two arrays: in the first, we list the index of the terminating symbol at the corresponding leaf and in the second, we list the index at which the element was last seen. This permits us to reduce document retrieval to FIND-LESS-THAN.

To solve this, we first fill out the elements of $B$ by iterating through $B$ and, for $1 \leq e \leq |T| + 1$, storing into $B[e]$ the index, $f$, of the last occurrence of $A[e]$ (if this is the first time we are seeing the element $A[e]$, then store $-1$ into $B[e]$). This process is illustrated in Figure 7. The idea is that at each element in $B$, we store the last time the corresponding element in $A$ was seen. Our problem of finding the distinct $d$ is then equivalent to finding all of the elements in $B$ between $i$ and $j$ that have a value less than $i$. The justification is as follows: assume $i \leq e \leq j$. If $B[e] \leq i$, then, within the $[i{:}j]$ interval, $A[e]$ has not been seen (since the element was last seen outside the interval). If $B[e] \geq i$, then it means that $A[e]$ was last seen within this interval and so the contents of $A[e]$ do not represent a newly seen index.

So now we have reduced document retrieval to **FIND-LESS-THAN**, which we define as *finding the list of elements in a given interval* $[i{:}j]$ *that are less than a pre-specified value,* $s^*$. As the previous paragraph implies, we will use FIND-LESS-THAN by taking our interval to be $B[i{:}j]$, and $s^*$ to be $i$, the index demarcating the left-hand side of the interval.

To solve FIND-LESS-THAN, we reduce it to Range Minimum Query (RMQ). Assuming we have an RMQ primitive, we can solve FIND-LESS-THAN as follows:

- use RMQ to return the index, $k$, of the minimum value, $v$, in the range $[i{:}j]$
- if $v \geq s^*$, then return
- if $v < s^*$, then:
  - output $v$
  - recursively call FIND-LESS-THAN with the interval equal to $[i{:}k-1]$
  - recursively call FIND-LESS-THAN with the interval equal to $[k+1{:}j]$

The running time of FIND-LESS-THAN is clearly $O((\text{\# of unique indices})\cdot(\text{cost of RMQ}))$, and the correctness derives from the fact that we are picking out all of the elements less than $s^*$, starting from the minimum element in the interval.

We will discuss RMQ below; before doing so, let's review how we got here. We started with the document retrieval problem and noted that a suffix tree would be helpful as a data structure but that, for document retrieval, we cannot naively use the suffix tree as we did for string matching. Rather, what we wanted was an efficient way to extract the unique elements from the leaves of a particular subtree of the suffix tree. To solve this problem, we did two things: we first created arrays $A$ and $B$ that stored our elements and the index at which they were last seen. Second, we then demonstrated an algorithm, FIND-LESS-THAN, that takes the array $B$ as input and outputs a list of unique documents. FIND-LESS-THAN relied on RMQ, so now we wish to solve RMQ, to which we now turn our attention.

## 6 Range Minimum Query and Least Common Ancestors

This portion of the lecture is derived from a paper by Gabow, Bentley and Tarjan [GBT84]. The problem we are trying to solve is: given an array $A$ and an interval $[i{:}j]$, output the index, $k$ (where $i \leq k \leq j$), of the minimum element in the interval. Below we outline the solution.

## 6.1 Cartesian Tree construction

We first pre-process the array to construct the **Cartesian Tree** of the array as follows:

- the root of the Cartesian Tree is the minimum element of the array
- the left and right subtrees are the recursive Cartesian Trees of the left and right pieces of the array, where we get left and right pieces by partitioning the array into two pieces around the root element (each section corresponds to the set of indices less than or greater than the index of the root element).

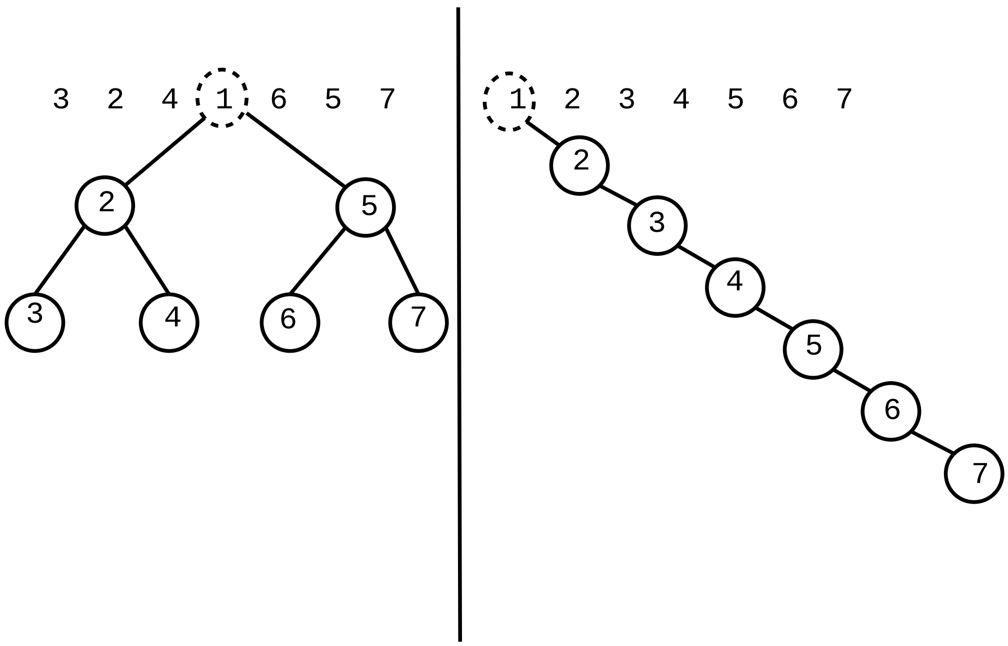


Figure 8: Cartesian Trees constructed from arrays

Figure 8 shows two different Cartesian Trees for two different arrays.

Now that we have a Cartesian Tree, we can answer questions about the minimum in any interval $[i{:}j]$: *the minimum is simply the least (or lowest) common ancestor of the two nodes in the Cartesian Tree.* This is intuitive from the construction; for a full proof see the paper by Harel and Tarjan [HT84]. We now examine the LCA problem.

## 6.2 Least (Lowest) Common Ancestor

In the next lecture, we will see how to pre-process the array in linear time with linear space so that we can find the Least Common Ancestor (LCA) in constant time[3]. For now, we assume that LCA is a primitive that returns in constant time. We will shortly illustrate some of the power of the LCA primitive. For clarity, we quickly highlight the series of reductions that led us to LCA.

[3]The main idea is to reduce the LCA problem to an easier version of RMQ, "RMQ Lite". This series of reductions is not circular because we solve the "RMQ Lite" problem without relying on LCA.

**Reductions:** We have arrived at LCA via a series of reductions, summarized here: DOCUMENT-RETRIEVAL $\Longrightarrow$ FIND-LESS-THAN $\Longrightarrow$ RMQ $\Longrightarrow$ LCA[4].

## 6.3 Applications of LCA to Suffix Trees

We finish up the lecture by showing how the LCA primitive[5] applied to suffix trees results in solutions to several problems related to string matching.

We first note that when LCA is applied to a trie, it returns the longest common prefix (LCP) of two strings (since the lowest common ancestor represents a common path, or series of letters, on the way to the two strings). Moreover, when LCA is applied to a suffix tree for text $T$, we get the LCP of 2 substrings of $T$. And if the suffix tree is the suffix tree for multiple texts $T_1, T_2, \ldots T_k$, *the LCA algorithm determines the LCP of substrings that originate in possibly different texts.*

### 6.3.1 Finding the Longest Palindrome

We use this last fact to determine, in time $O(|T|)$, the longest palindrome that occurs in a text $T$. This requires that we do the following pre-processing:

1. Create a suffix tree $S$ that consists of the suffixes of $T$ and $R(T)$, where $R(T)$ denotes the reverse of text $T$
2. Pre-process the tree by assigning to each node a "pattern depth", representing the number of letters on the path from the given node to the root
3. For each $i$, create a pointer to the node that represents $T[i{:}]$ and $R(T)[i{:}]$. In other words, we store a mapping that takes us from suffixes of $T$ and $R(T)$ to the corresponding nodes in the suffix tree
4. Do the pre-processing to support LCA queries

Once we have done this pre-processing, we iterate, taking $i$ from 1 to $|T|$ and use LCA to identify, in constant time, the LCP of:

- $T[i{:}]$ and $R(T)[|T| - i{:}]$
- $T[i{:}]$ and $R(T)[|T| - i + 1{:}]$

which is correct since these two comparisons represent the possibilities for palindromes centered at $i$ or in between $i$ and a neighbor. As we execute LCA for each $i$, we store a pointer to the longest palindrome we have seen so far (and we know how long each one is, since we have pre-computed the pattern depths). Since LCA is constant time, the whole procedure takes time $O(|T|)$.

[4] And LCA $\Longrightarrow$ "RMQ Lite", which we will discuss in the next lecture

[5] When we talk of the LCA primitive, we mean the algorithm that pre-processes a tree and then, in response to queries, returns a pointer to the least common ancestor in constant time.

### 6.3.2 DNA Application

One application of the above solution is determining where a strand of DNA might fold. DNA consists of a string of nucleotides, each of which is, for the purposes of this discussion, distinguished by an attached base. There are four types of base: thymine (T) and adenine (A), which bond to each other; and cytosine (C) and guanine (G), which bond to each other. If we know the longest "palindrome" of bonded pairs, we might be able to tell where the DNA will fold, that is, where the kinks might be. For example, the string of DNA ....$ATCCGGAT$.... might fold in such a way that the $ATCC$ piece bonds to the $GGAT$ piece (picture folding the strand over in between the middle $C$ and the middle $G$).

To determine the longest "palindrome" of bonded pairs, simply use the longest palindrome algorithm and data structure, above. The only change is that instead of making a suffix tree out of $T$ and $R(T)$, the suffix tree $S$ should be constructed from suffixes of $T$ and $R(\overline{T})$, where $\overline{T}$ is the string with the elements of $T$ replaced with their bonding complements. A longest prefix match in these two strings indicates where the longest string of "palindromic" bonding pairs occurs.

### 6.3.3 Searching for $P$ in $T$ with $\leq k$ mismatches

The final problem we examine is finding an "occurrence" of $P$ in $T$ with $k$ or fewer mismatches. After pre-processing, we can do this in $O(k \cdot |T|)$ time, as follows:

1. Construct a suffix tree $S$ out of the suffixes of $T$ and $P$
2. For each $i$, create a pointer to the node that represents $T[i{:}]$
3. Do the pre-processing to support LCA queries
4. For $i$ from 1 to $|T|$:
   - determine the LCP of $T[i{:}]$ and $P$
   - if the two strings match fully, then we're done
   - if there is a mismatch, the LCP will "stop" at the mismatch. If the mismatch occurs at, for example, position $p$ in $P$, then we keep going, only now we are comparing $T[i + p{:}]$ and $P[p{:}]$. By calling the LCP algorithm again, we can find the next mismatch. We can keep doing this $k$ times, each time advancing our index $p$.

In the worst case, we require $k$ calls to LCP for each $i$, so this algorithm runs in time $O(k \cdot |T|)$, since each call to LCP takes constant time, after pre-processing.

## References

[BM77] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[GBT84] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[GK97] R. Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19(3):331–353, 1997.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, March 1987.

[McC76] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666, San Francisco, CA, January 2002.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.