15-451/651: Design & Analysis of Algorithms November 21, 2013
Lecture #26 last changed: December 11, 2013

There's an entire field dedicated to solving problems on strings. The book "Algorithms on Strings, Trees, and Sequences" by Dan Gusfield covers this field of research. (Not to mention the algorithms in this lecture.) And you can also check out the course 02-714: String Algorithms here at CMU.

Here are some sample problems:

- Given a text string and a pattern, find all occurrences of the pattern in the text. (Classic text search)
- The above problem where the pattern can have "don't cares" in it.
- Given a string, find longest string that occurs twice in it.
- Compute the "edit" distance between two strings, where various editing operations are defined, along with their costs.
- Given a set of strings, compute the cheapest tree that connects them all together (phylogeny tree)
- Compute the shortest "superstring" of a set of strings. That is the shortest string that contains all the given strings as a substring. (Important in sequencing.)

Not only do these problems arise in our everyday lives (how do the `grep` and `diff` algorithms work?), but they also have many applications in computational biology. We're only going to scratch the surface of this field and talk about a couple of these problems and algorithms for them.

1 The Knuth-Morris-Pratt Algorithm

This algorithm can solve the classic text search problem in linear time in the length of the text string. (It can also be used for a variety of other string searching problems.) Formally, you have a pattern P of length p, and a text T of length t, and you want to find all locations i such that $T[i \ldots, i+p-1] = P$. For example, if you have a pattern $P =$ `ana` and $T =$ `banana`, then you would want to output locations $\{1, 3\}$, since $T[1 \ldots 4] = T[3 \ldots 6] =$ `ana`.[1]

An easy solution to this problem takes time $O(p \cdot t)$. This is fine if p is small, but as p gets large this becomes bad. *Can we do better?* It seems that if once we have checked whether $T[i \ldots i+p-1]$ matches P or not, we have a lot of information that we can use to determine if $T[i+1 \ldots i+p]$ gives a match. How do we do this?

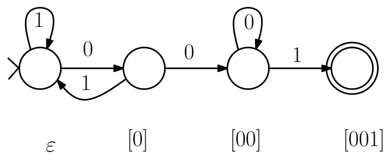
Any ideas?

One high-level solution to this problem is to build a deterministic finite automaton M_P, which depends on the pattern. It consists of $p = |P|$ states, which you should think of as arranged in a line. We feed in the text T to this automaton. The properties of the DFA M ensure that we're in the j^{th} state if and only if at the current location in the string we've already matched a sequence of j characters from the pattern. Now we compare the next two characters. If we get a match we move to the next state in the list (matching $j+1$ characters). If we get a mismatch, we can skip back to some previous state. Which one? The start state of the automaton? Nope, that would be

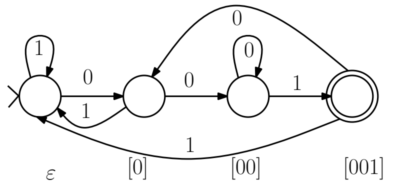
[1]We will assume that the strings are from some alphabet Σ. In some cases we will also assume that $|\Sigma|$ is a constant, we'll mention when we're doing so.

wrong. We go to the furthest back state that we know we can go to based on the information that we have.

For example, if the pattern $P = \texttt{001}$ and suppose we consider the DFA:

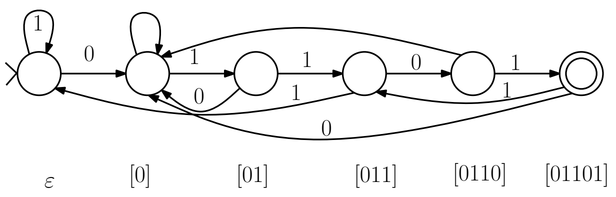


We would reach the final state (the one with the double circle) exactly when we have seen the pattern. This would allow us to find the first occurrence of pattern P in the text T. And in order to find all the occurrences of P, we could alter the DFA slightly to get this one:



We could output the current location in T every time we hit the final state, and then output all occurrences of P in the text T.

Another example: if $P = 01101$ then we'd build



Clearly, once we have the DFA M_P, it takes just $O(t)$ time to feed the text T to the DFA, and record the locations in T when we reach the final state. If it takes us $f(P)$ time to build the DFA, the total run-time of the search algorithm would be $f(P) + O(t)$. It is easy to construct the DFA in time $O(p^3|\Sigma|)$, where recall that Σ is the alphabet. However, the algorithm of Knuth, Morris, and Pratt [2] suggests a way to compute the DFA in only $O(p|\Sigma|)$ time. The DFA is of size $O(p|\Sigma|)$, so this is optimal. In fact, their algorithm goes even further. While it is not presented as such, it can be viewed as building a related DFA-like automaton of size $O(p)$ (independent of the size of alphabet size Σ) that can be used for the string matching problem in $O(t)$ time.

But we won't cover this today: if you are interested, check out Danny Sleator's notes for 15-451 from Spring 2013. (Or depending on time, we might do this on the board.)

[2] The historical notes in the original paper are interesting to read. Also, the *dramatis personae* may be familiar to you: Jim Morris is a professor of computer science here at Carnegie Mellon. He was Dean of the School of Computer Science from 1999-2004. Both Don Knuth and Vaughan Pratt are professors of computer science at Stanford. Don Knuth's books *The Art of Computer Programming* have been super-influential for the field, and he also developed the TeX typesetting system which has been used to create this document. He won the Turing Award in 1974. Vaughan Pratt was one of the authors of the deterministic median finding algorithm (along with Manuel Blum, Bob Floyd, Ron Rivest and Bob Tarjan); he also gave the proof you saw that PRIMES is in NP.

2 The Karp-Rabin Algorithm (a.k.a. the "Fingerprint" Method)

A different approach to the classic string matching algorithm is to use randomization. This solution is due to Karp and Rabin.[3]

The idea here is to design a special kind of "rolling" hash function $h(S)$ on strings S that has the following properties:

1. After a small amount of precomputation, given the hash of the string $T[i \ldots (i+p-1)]$, we can compute $h(T[(i+1) \ldots (i+p)])$ with a constant number of arithmetic operations. (All our operations will be done modulo some prime number $q = O(t \cdot p^2)$.)

2. The probability of a collision will be "low". We'll be more precise later.

So ignoring the collisions, which have "low" probability, it's easy to see how this hash function can solve the classic text search problem. We first compute $h(P)$ the hash function of the pattern. Then try all substrings $T[i \ldots (i+p-1)]$ for all $i \in \{0, 1, \ldots, t-p\}$ and see which ones of them have hash value equal to $h(P)$, and output all locations where $h(T[i \ldots (i+p-1)]) = h(P)$. [4] There are $t-p+1$ such strings of suitable length, so the algorithm will take time $O(t-p+1)$ for these tests, $O(p)$ to compute $h(P)$ and for the preprocessing, so overall $O(t+p)$ time.

For simplicity, assume that the alphabet $\Sigma = \{0, 1\}$. The Karp-Rabin hash family is the following. Choose a uniformly random prime $q \in \{2, \ldots, K\}$ for some parameter K to be chosen later. (How do we choose a random prime? We will talk about this later.) Now, define

$$h_q(T[i \ldots j]) = (T[i] \cdot 2^{j-i} + T[i+1] \cdot 2^{j-i-1} + \ldots + T[j] \cdot 2^0) \bmod q$$

That is, interpret the $(j-i)$ bits in the substring as a number written in binary, and take the residue modulo q. This is often called the "fingerprint" of the substring.

Good. Now what is the hash of the "next" location, if we were to move the "window" over by 1? Just plug in the definition

$$h_q(T[i+1 \ldots j+1]) = (T[i+1] \cdot 2^{j-i} + T[i+2] \cdot 2^{j-i-1} + \ldots + T[j+1] \cdot 2^0) \bmod q$$

which is the same as

$$h_q(T[i+1 \ldots j+1]) = \left(\left(h_q(T[i \ldots j]) - T[i] \cdot 2^{j-i}\right) \cdot 2 + T[j+1] \cdot 2^0\right) \bmod q$$

Now, suppose we are given $h_q(T[i \ldots (i+p-1)])$ and want to compute $h_q(T[(i+1) \ldots (i+p)])$. What do we do?

Easy. Take $h_q(T[i \ldots (i+p-1)])$, subtract $T[i] \cdot 2^{p-1} \bmod q$ from it (note that $T[i] \cdot 2^{p-1} \bmod q$ is either 0 or $2^{p-1} \bmod q$, and we can precompute and store $2^{p-1} \bmod q$), then multiply by 2, and add in $T[j+1]$. (All modulo q, of course.) A constant number of arithmetic operations modulo q, as advertised.

[3] Again, familiar names. Dick Karp is a professor of computer science at Berkeley, and won the Turing award in 1985. Among other things, he developed two of the max-flow algorithms you saw, and his 1972 paper showed that many natural algorithmic problems were NP complete. Michael Rabin is professor at Harvard; he won the Turing award in 1976 (jointly with CMU's Dana Scott). You may know him from the popular Miller-Rabin randomized primality test (the Miller there is our own Gary Miller); he's responsible for many algorithms in cryptography.

[4] Note this is a Monte-Carlo algorithm, since the algorithm may make a mistake and output some location i such that $h(T[i \ldots (i+p-1)]) = h(P)$ but $T[i \ldots (i+p-1)] \neq P$. We will show the probability of this is small.

2.1 Probability of False Positives

What about the possibility of false matches? Suppose for any fixed location i, the probability of an incorrect match is δ. Then by a union bound over $t-p+1$ locations we perform the equality test, the probability of outputting some false positive is at most $(t-p+1)\cdot\delta \leq t\delta$. If we want the final probability of error to be at most $1/2$, we should ensure that $\delta \leq 1/(2t)$. How?

Notice the only randomness is in the choice of the random prime q. We make a mistake when the number represented by the p-bit substring $T[i\ldots(i+p-1)]$ (call this number a) is *not equal* to the number represented by the p-bit pattern P (call this second number b), but these two numbers are the same modulo the random q (i.e., $a \equiv_p b$). By the definition of being equivalent modulo q, this means that q divides $|a-b|$. Now $|a-b|$ is also a p-bit number, so it can have less than p distinct prime divisors. (Each prime divisor is at least 2, and $|a-b| < 2^p$.) And if $q|(a-b)$, then q must have been one of these "bad" values, these prime divisors.

We would like to claim that choosing a uniformly random prime number q in the range $\{2,\ldots,K\}$, the chance that we choose one of (at most) p bad values is smaller than $1/(2t)$. For this, it suffices to choose K large enough such that there are at least $2pt$ primes between 2 and K. For this we use the Prime Number theorem: if there are $\pi(x)$ primes between 0 and x, then the theorem says that $\lim_{n\to\infty}\frac{\pi(x)}{x/(\ln x)} = 1$. And while this is just an asymptotic statement, we also know that $\pi(n) \geq \frac{7}{8}\frac{n}{\ln n}$ (this was proved by Chebyshev back in 1848). Now setting K to equal, say, $10pt\ln pt$ ensures that $\pi(K) \geq 2pt$ for large enough pt, which proves the result.

If you want to reduce the error probability, you could either pick several independent primes q and perform the string matches in parallel (claiming that there is a match at location i only when all the fingerprints match), or you could increase the value of K, which increases the range from which you pick your random prime.[5]

2.2 Picking a Random Prime

One detail is how to pick a random prime in some range $\{0,1,\ldots,M\}$. The answer is easy.

- Pick a random integer X in the range $\{0,1,\ldots,M\}$.
- Check if X is a prime. If so, output it. Else go back to the first step.

How would you pick a random number in the prescribed range? Just pick a uniformly random bit string of length $\lfloor\log_2 M\rfloor + 1$. (We assume we have access to a source of random bits.) If it represents a number $\leq M$, output it, else repeat. The chance that you will get a number $\leq M$ is at least half, so in expectation you have to repeat this process at most twice.

How do you check if X is prime? You can use the Miller-Rabin randomized primality test (which may err, but it will only output "prime" when the number is composite with very low probability). There are other randomized primality tests as well, see the Wikipedia page. Or you can use the Agrawal-Kayal-Saxena primality test, which has a worse runtime, but is deterministic and hence guaranteed to be correct.

Finally, how many repetitions would you have to do in the above algorithm before you output a prime? Again, you can appeal to the Prime Number Theorem. The number of primes in the

[5]Converting this to a Las Vegas test is a little more complicated. Clearly if the algorithm doesn't output location i, we know that $T[i\ldots(i+p-1)]$ does not match P. But checking if the locations that were output were bonafide matches might take a lot of time: imagine a text containing t copies of a, and a pattern with p copies of a. Naïvely checking for false matches might take $O(pt)$ time. There are ways to check in linear time if there exists a false match in the output locations which are not difficult; see the book for details.

range $\{0, 1, \ldots, M\}$ is $\Omega(\frac{M}{\ln M})$, so a random number in this range is prime with probability $\Omega(\frac{1}{\ln M})$. Hence the expected number of repetitions is $O(\log M)$.

3 Suffix Trees

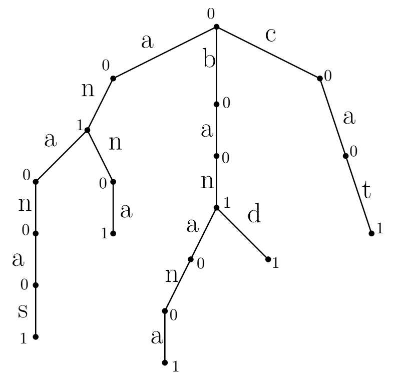
Consider a string T of length t (long). Our goal is to preprocess T and construct a data structure, to allow various kinds of queries on T to be done efficiently. The most basic example of which is simply this: given a pattern P of length p, find all occurrences of P in the text T. What is the performance we aiming for?

- The time to find all occurrences of pattern P in T should be O(p + k) where k is the number of occurrences of P in T.
- Moreover, ideally we would require $O(t)$ time to do the preprocessing, and $O(t)$ space to store the data structure.

Suffix trees are a solution to this problem, with all these ideal properties.[6] They can be used to solve many other problems as well. In this lecture, we'll consider the alphabet size to be $|\Sigma| = O(1)$.

3.1 Tries

The first piece of the puzzle is a *trie*, a data structure for storing a set of strings. This is a tree, where each edge of the tree is labeled with a character of the alphabet. Each node then implicitly represents a certain string of characters. Specifically a node v represents the string of letters on the edges we follow to get from the root to v. (The root represents the empty string.) Each node has a bit in it that indicates whether the path from the root to this node is a member of the set—if the bit is set, we say the node is marked.



banana
ananas
ban
band
anna
cat
an

Since our alphabet is small, we can use an array of pointers at each node to point at the subtrees of it. So to determine if a pattern P occurs in our set we simply traverse down from the root of the tree one character at a time until we either (1) walk off the bottom of the tree, in which case P does not occur, or (2) we stop at some node v. We now know that P is a prefix of some string in our set. And if v is marked, then P is in our set, otherwise it is not.

This search process takes $O(p)$ time because each step simply looks up the next character of P in an array of child pointers from the current node. (We used that $|\Sigma| = O(1)$ here.)

Note that if we also keep a count at each node v of the number of marked nodes in the subtree rooted at v, we can then efficiently determine for a pattern P, how many members of my set of strings begin with the characters of P. (Not shown in the example above, but you can get the count at v by the number of 1s in its subtree.)

[6]Suffix trees were invented by Peter Wiener.

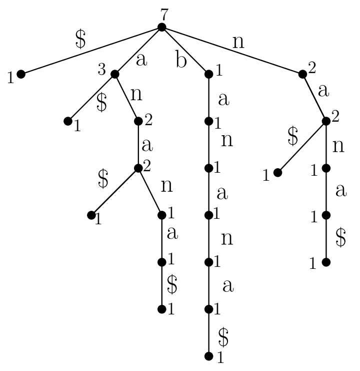
3.2 Returning to Suffix Trees

Our first attempt to build a data structure that solves this problem is to build a trie which stores all the strings which are suffixes of the given text T. It's going to be useful to avoid having one suffix match the beginning of another suffix. So in order to avoid this we will affix a special character denoted "\$" to the end of the text T, which occurs nowhere else in T. (This character is lexicographically less than any other character.)

For example if the text were $T =$ `banana$`, the prefixes of T are then

```
banana$
anana$
nana$
ana$
na$
a$
$
```

(Since we appended the \$ sign to the end, we're not including the empty prefix here.) And the trie with counts as described above would be the one below:

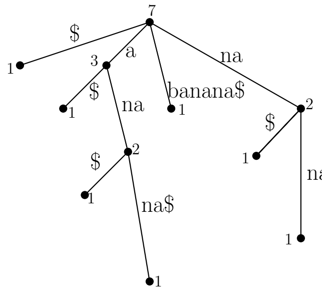


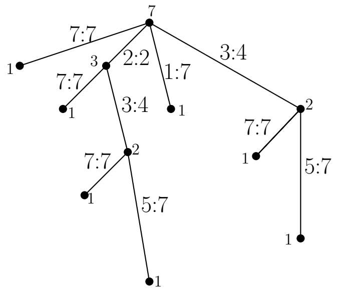
Suppose we have constructed the trie containing all these suffixes of T (and the counts). Now given a pattern P, we can count the number of occurrences of P in T in $O(|P|)$ time: we just walk down the trie and when we run out of P we look at the count of the node v we're sitting on. It's our answer.

But there are a number of problems with this solution. First of all, the space to store this tree (as described) could be as large as $\Theta(t^2)$. Also, it's unsatisfactory in that it does not tell us where in s these patterns occur. Finally, it will also take too long to build it.

The first two issues are easy to handle: we can create a "compressed" tree of size only $O(t)$. Since no string occurs as a prefix of any other, we can divide the nodes of our trie into *internal* and *leaf* nodes. The leaf nodes have no children, and represent a suffix of T. So we can have each leaf node point to the place in T where the given suffix begins.

Moreover, if there is a long path in the trie with branching factor 1 at each node on that path. We can compress that path into a single edge that represents the entire path. Moreover, the string of characters corresponding to that path must occur in T, so we can represent it implicitly by a pair of pointers into the string T. So an edge is now labeled with a pair of indices into T instead of just a single character (or a string). Here's the example (with the substrings labeling the edges on the right, and the start-end pairs labeing them on the right):





This representation uses $O(t)$ space. (We count pointers as $O(1)$ space.) Why? Each internal node now has degree at least 2, hence the total number of nodes in the tree is at most twice the number of leaves. (Exercise: prove this!) But each leaf corresponds to some suffix of T, and there are t suffixes.

What about the time to build the data structure. Let's first look at the naïve construction, by adding suffixes into it one at a time. To add a new suffix, we walk down the current tree until we come to a place where the path leads off of the current tree. (This must occur because the suffix is not already in the tree.) This could happen in the middle of an edge, or at an already existing node. In the former case, we split the edge in two and add a new node with a branching factor of 2 in the middle of it. In the latter case we simply add a new edge from an already existing node. In either case the process terminates with a tree containing $O(t)$ nodes, and the running time of this naive construction algorithm is $O(t^2)$.

3.3 Other Applications of Suffix Trees

There are many other applications of suffix trees to practical problems on strings. Gusfield discusses many of these in his book. We'll just mention just one here.

Longest Common Substring of Two Strings. Given two strings S and T, what is the longest substring that occurs in both of them. For example if $S =$`boogie` and $T =$`ogre` then the answer is `og`. How to compute this efficiently? The answer is to use suffix trees. Here's how.

Construct a new string $U = S\%T$. That is, concatenate S and T together with an intervening special character that occurs nowhere else (indicated here by "%"). Let n be the sum of the lengths of the two strings. Now construct the suffix tree for U. Every leaf of the suffix tree represents a suffix that begins in S or in T. Mark every internal node with two bits: one that indicates if this subtree contains a substring of S, and another for T. These bits can be computed by depth first search in linear time. Now take the deepest node in the suffix tree (in the sense of the longest string in the suffix tree) that has both marks. This tells you the the longest common substring.

This is a very elegant solution to a natural problem. Before suffix trees, an algorithm of Karp, Miller, and Rosenberg gave an $O(n \log n)$ time solution, and Knuth had even conjectured a lower bound of $\Omega(n \log n)$.

3.4 Computing the Suffix Tree

Let's see how to compute the suffix tree from two other constructs of the string T. They are the *suffix array* and the *prefix length array.*

Imagine that you write down all the suffixes of a string T. The i^{th} suffix is the one that begins at position i. Now imagine that you sort all these suffixes. And you write down the indices of them in an array in their sorted order. This is the suffix array. For example, suppose $T =$ `banana$` (as above) and we've sorted the suffixes.

```
    0  1  2  3  4  5  6
    b  a  n  a  n  a  $

 6:  $
 5:  a$
 3:  ana$
 1:  anana$
 0:  banana$
 4:  na$
 2:  nana$
```

The numbers to the left are the indices of these suffixes. So the *suffix array* is:

```
6 5 3 1 0 4 2
```

(If you were to do an in-order traversal of the suffix tree, you'd get these suffixes in this order, since you'd reach the corresponding leaves in sorted order.)

Now each successive suffix in this order matches the previous one in some number of letters. (Maybe zero letters.) This gives the common prefix lengths array. In this case we have:

```
suffix array is:               6 5 3 1 0 4 2
common prefix lengths array     0 1 3 0 0 2
```

First, let's show how given these two arrays, the suffix tree can be computed in linear time. Then we'll talk about how to compute these arrays fast.

Arrays to Suffix Tree: We add the suffixes one at a time into a partially built suffix tree in the order that they appear in the suffix array. At any point in time, we keep track of the sequence of nodes on the path from the most recently added leaf to the root. To add the next suffix, we find where this suffix's path deviates from the current path we're keeping track of. To do this, we just use the common prefix length value. We walk up the path until we pass this prefix length. This tells us where to add the new node. The time to build the suffix tree in this fashion is the same as the time it takes to traverse the suffix tree from left to right. That is, it's linear time, time $O(t)$ for a text of length t.

Computing these Arrays There are linear time algorithms for this problem, but here let us give a randomized method that takes time $O(t \log^2 t)$ based on Karp-Rabin fingerprinting. The observation is that if we could compare two suffixes in $O(1)$ time we could just sort them in $O(t \log t)$ time to get the suffix array. We'll show how to compare two suffixes in $O(\log t)$ time, which will also give the prefix lengths array as a by-product.

Using Karp-Rabin fingerprinting we can compare two strings for equality in $O(1)$ time (with a tiny error probability).[7] To compare two suffixes a and b to determine their lexicographic ordering, we use binary search to find the shortest length r such that $a[0 \ldots (r-1)] = b[0 \ldots (r-1)]$, but $a[0 \ldots r] \neq b[0 \ldots r]$. Then $a < b$ exactly when $a[r] < b[r]$. Furthermore this also tells us the common prefix length between the two suffixes.

4 Summary

We saw different approaches to exact string matching problems: given a pattern P and a text T, find all occurrences of P within T. We saw:

[7] Really, we're counting the number of arithmetic operations modulo a prime p, where p is polynomial in t.

- the deterministic Knuth-Morris-Pratt algorithm which runs in time $O(t + p)$. Here if you have a pattern you want to find in many texts, you can preprocess the pattern in $O(p)$ time and space, and then search over multiple texts, the search in text t_i taking the time $O(t_i)$.
- The randomized Karp-Rabin fingerprinting scheme which also has a similar running time. However, it is a versatile idea and extends to different settings (like 2-dimensional pattern matching).
- The suffix-tree construction. Here you preprocess the text in $O(t)$ time and space, and then you can perform many different operations (including searching for different patterns) in time $O(p + n_{p,t})$, where $n_{p,t}$ is the number of occurrences of pattern p in text t.

Each one its advantages. And in recitation yesterday, you saw an approach to pattern matching with “don’t-cares”. There’s a huge amount more to learn in this area, see the book and advanced courses for more details.