Cool String Tricks

Alex Chen

November 18, 2011

1 Introduction

Strings are pretty cool! Thus, we need some cool string tricks to deal with them. String problems are present in all levels of USACO, and are one of the topics most seen outside USACO (in the *real* computer science world, from web applications to bioinformatics). Thus, string processing and manipulation algorithms are definitely worth learning.

What is a string? Most simply, a **string is an array of characters**. For example, "albert" is a string made up of the characters 'a', 'l', 'b', 'e', 'r', and 't'. For this lecture, all strings will be in double quotes (" ") and all characters will be in single quotes (' '). Java and C++ both use strings in this form, so it is something to get used to. One important note is that a string's characters must come from a *finite* alphabet. As an example, the English alphabet contains 52 characters (26 different letters).

The length of a string is merely the number of characters. "albert" is a length 6 string. Characters of strings can also be referred to through indexing. These characters are numbered starting from 0 (just like a regular array). Two strings are equal if they have the same length and contain the same characters in all indices. In this lecture, we will refer to the i^{th} character of string S by S_i.

2 String Searching

2.1 Problem

The string searching problem is one of the most common string problems. Given string S, which has length n, and string T, which has length m, find whether T is a *substring* of S. That is, decide whether there is some set of m consecutive characters in S that forms a string equal to T. For example, "lber" is a substring of "albert" but "alert" is not a substring of "albert".

2.2 Naive Solution

We can check the equality of two length m strings in $O(m)$ time by comparing each character one by one (there is a low constant factor associated with this, if implemented correctly). There are a total of $n-m+1$ substrings of length m to check in S. Thus, the total runtime of a brute force algorithm, comparing all possible substrings of S, with correct length, to T, runs in $O(m \times (n-m+1)) = O(nm - m^2 + m) = O(nm)$, under the assumption that $n \geq m$. Note that this assumption is safe, because if $n < m$, then T cannot possible be in S. $O(nm)$ is pretty slow, however. Java's substring search uses this. We can do better.

2.3 Other Solutions

We can solve this problem in linear (!) time ($O(n+m)$) using some tricky algorithms. These are pretty complex but feel free to look them up on your own. For now, they will probably be the subject of a future lecture. Linear time solutions include:

- suffix trees

- Knuth-Morris-Pratt string matching algorithm
- Boyer-Moore string search algorithm
- and more!

3 Rabin-Karp String Hash

Among efficient string search algorithms, Rabin-Karp hashing is by far the easiest to implement and easiest to use. It is also very versatile, being applicable beyond strings. It conveniently solves the string hashing problem in worst case $O(n+m)$ time with $O(n)$ preprocessing time. It does this by allowing the comparison of unequal substrings in constant time.

3.1 Hash Function

The Rabin-Karp hash function *hashes* a string using a polynomial function. If you are not familiar with hashing, it is simply the conversion of some data type (here, a string) into a simpler one (here, an integer). By comparing strings to integers, we can compare them in constant time by checking equality of their representative integers. A good hash function dictates that two equal strings must have the same hash values. Given a string S of length n, we let the hash function be

$$H(S) = \sum_{i=0}^{n-1} S_i \cdot p^i = S_0 + p \cdot S_1 + p^2 \cdot S_2 + \cdots + p^{n-1} \cdot S_{n-1},$$

where p is some arbitrary prime. For this expression we can replace S_i with its integer equivalent. This hash function works well because it is very input sensitive. Two strings that differ by just one character might have very different hash values. Often, the value of this hash function might be very high. Thus, we can take all values modulus some very large prime (or we can just let integer overflow modulus the expression for us, but that is not as reliable).

3.2 String Comparison

If $H(S) = H(T)$, then $S = T$ with a high probability. Usually, to check whether two strings are actually equal after checking their hashes, we compare them character by character just in case. However, because usually very few substrings of S are equal to T, character by character comparison does not occur very often (the rest all fail the first test, the hash value test).

However, this only lets us compare two strings and not necessarily their substrings. To fix this, we can do something clever that is related to prefix sums. Let us define

$$H(S, j) = \sum_{i=0}^{j-1} S_i \cdot p^i,$$

which is merely the hash value of the substring of S containing the first j characters. This lets us find a representation of the hash value of any substring of S using subtraction:

$$H(\text{substring of } S \text{ from index } j \text{ to index } k) = \frac{1}{p^k}(H(S, j+1) - H(S, k)).$$

Note that we must divide by p^k to get rid of that extra factor present. Using precomputed values for $H(S, j)$, we can calculate the hash value of any substring in $O(1)$ time! Note that this precomputation is linear: we can calculate $H(S, j)$ from $H(S, j-1)$ because $H(S, j) = H(S, j-1) + p^{j-1} \cdot S_{j-1}$.

3.3 Something Tricky

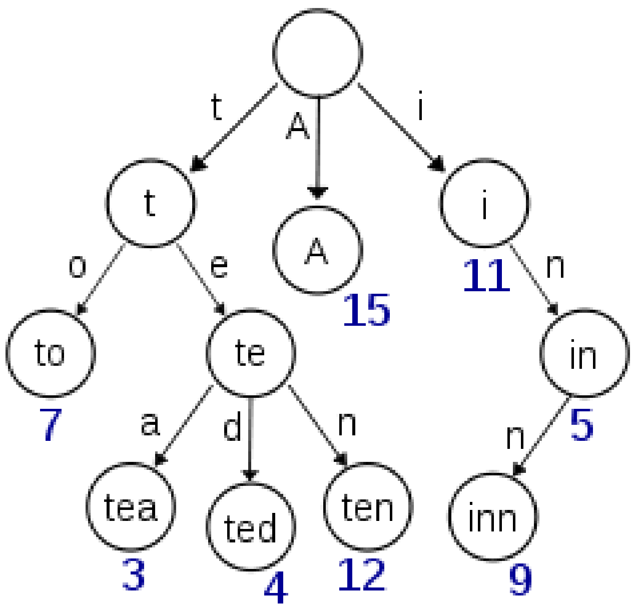
When we hash T, we use a polynomial function that begins at the 0^{th} power of p. When we hash a substring of S, we have to divide by some power of p to get the polynomial function to begin at the 0^{th} power of p. However, in modulus world, we cannot divide by a number because a lot of factors might have been lost in the modulus process. There is an easy fix, however.

Instead of dividing the substring of S by a power of p, we merely multiply the hash value of T by a power of p such that they start at the same power. For example, if the hash value of the substring of S starts with p^5, we can multiply the hash value of T by p^5 because the polynomial for $H(T)$ begins with $p^0 = 1$. Then, comparison works!

How is multiply by some power of p still constant time? In order to make this constant, we must precompute not only the hash values of prefixes S and T, but we must also precompute and store power of p for quick access.

4 Tries

A very useful data structure for strings is a *trie*. A trie is merely a tree used to store many strings.



The words in this trie are: “A”, “to”, “tea”, “ted”, “ten”, “i”, “in”, and “inn” (source of image: Wikipedia).

The root itself is a dummy node. Each path from the root to a leaf spells a word (and sometimes, a path from the root to an internal node also spells a word). A trie is useful for storing a dictionary. As an exercise for you, think about how you can create a trie of all the words in a single dictionary in time proportional to the total number of characters.

A trie can also be used to sort a dictionary in linear time. Consider a certain traversal of the trie.

5 Problems

1. Given 1000 strings of length 1000 each, find, for each pair of strings, the length of the longest shared prefix of the strings. For example, “albert” and “alert” have a longest shared prefix of 2 (“al”).

2. Under the assumption that Rabin-Karp hashing produces no collisions (do not make this assumption in real life), find the number of times a string of length 1000 occurs as a substring in a string of length $100,000$. *Bonus: Do this without storing any arrays other than the strings themselves. This means that precomputing and storing powers of* p *is not allowed!*

3. Given a string of length $100,000$, find the length of the longest palindromic substring. A palindrome is a string that equals its reverse. *Bonus: Do this in linear time. This uses a simple algorithm, but nothing specifically from this lecture.*

4. You are given a dictionary of words and many other strings. For each string, determine whether it is in the dictionary. You must do this in linear time (proportional to the total number of characters in the input words combined).

5. You are given a dictionary of words and many short strings. For each short string, determine the number of dictionary words that begins with it. Compute this in linear time (proportional to the total number of characters in the input words combined).

6. (USACO December 2010) You are given a string S of length $50,000$ and another string T of length $50,000$. You want to create T using substrings of S (substrings of S may overlap and be used more than once). Find the minimum number of substrings in S needed to create T.