Advanced String Algorithms

Spandan Das

April 2020

1 Introduction

Strings are one of the most basic data types, introduced to us early on when learning programming. Aside from basic string manipulation problems, many contest problems require the use of more advanced string algorithms. In this lecture, we'll cover dynamic programming with strings, string matching, and data structures for strings.

2 Dynamic Programming with Strings

There are a lot of interesting dynamic programming problems involving strings. We'll go over some of the most common ones that show up in competitions.

2.1 Longest Common Subsequence (LCS)

First, let's go over the definition of a subsequence. A subsequence is a sequence derived from another sequence by deleting some elements and maintaining the order of the others. A subsequence is NOT the same as a substring. For example, "apl" is a subsequence of "apple" (but not a substring), while "ppl" is both a subsequence and substring.

The LCS problem is defined as follows: given two strings, X and Y with lengths n and m, respectively, find the longest common subsequence between them. The naive method to solving this problem would be to generate all subsequences of X, check whether it is a subsequence of Y, and then return the longest subsequence found. This solution has a complexity of $O(m2^n)$.

We can develop a dynamic programming solution by noticing two properties. First, if X and Y end with the same element, then their LCS is the LCS of the sequences without the last element, with the common last element added to the end. Second, if X and Y end with different elements, then the LCS is the longer of $LCS(X_n, Y_{m-1})$ and $LCS(X_{n-1}, Y_m)$. It follows that our DP is:

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1})\hat{}\,x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Thus, our new and improved time complexity comes to $O(mn)$.

2.2 Longest Common Substring

Let's now look at a problem very similar to the previous problem. This time, given strings X and Y with lengths n and m, respectively, find the longest common *substring* between them. To find the solution, we again perform casework on the last elements of both strings. Just like before, if the last elements of the strings are the same, the LCSubStr is simply the longest common substring of the two strings without the last character plus 1. However, if the characters are not the same, then the DP table simply stores a 0. At the end of the process, we iterate through the DP table and the answer is the maximum value in the table. This makes our overall time complexity $O(mn)$.

We will revisit this problem later in the lecture when discussing string data structures.

2.3 Palindromic Subsequences

Given a string A with length n, find the longest palindrome that can be formed by deleting 0 or more characters (i.e. the longest palindromic subsequence, which we will abbreviate as LPS). For example, the LPS for "AABCDEBAZ" would be of length 5 (i.e. "ABCBA", "ABDBA", "ABEBA").

To find the DP solution, let's first try to find a recursion. Considering the structure of palindromes, the DP will most likely involve casework on the start and end elements of the string. The cases themselves are fairly easy to find. The first case is that the first and last elements of the string are the same. In this case, the LPS will simply be length of the longest palindrome without the first and last elements added to 2. The second case is that the start and end elements are different. In this case, the LPS will be the larger of the palindromic subsequences without the first or last character. All that remains to find are the base cases, so our final DP is

$$LPS(l,r) = \begin{cases} 1 & l = r \\ 2 & l+1 = r \texttt{ and } A[l] = A[r] \\ LPS(l+1, r-1) + 2 & A[l] = A[r] \\ max\{LPS(l+1,r), LPS(l,r-1)\} & A[l] \neq A[r]. \end{cases}$$

3 A Needle in a Haystack - String Matching

The string matching problem is defined as follows: given two strings - a text (of length n) and a pattern (of length m, $m \leq n$) - determine whether the pattern appears in the text. The problem is also known as "the needle in a haystack problem."

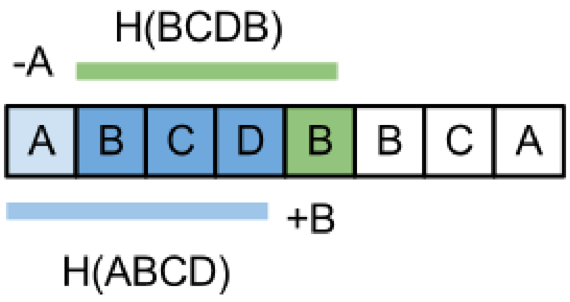
The naive method is obvious: for every position in the text, consider it a starting position of the pattern and see if you get a match. This has an overall time complexity of $O(mn)$. We will study two different algorithms which are better alternatives.

3.1 Rabin-Karp Algorithm

When trying to improve the naive solution, one idea that we may have is to calculate the hash of the pattern string and then compare it to the hash of the sliding window in the text. Comparing hash values is a very quick operation, since we just check if the two numbers (hash values) are equal. In the rare case that the hash values match, we can then go through and compare each character in the brute force manner. This method of string matching is known as the Rabin-Karp Algorithm.

3.1.1 Rolling Hash

The trick is to calculate the hash of the sliding window without having to iterate over each of its characters every time. To do this, we will use a technique known as **rolling hash**. Rolling hash uses the hash of the previous window and adjusts it to accommodate for the character that was "pushed out". The simplest implementation of a rolling hash would just be to add the numeric value of each character (i.e. ASCII value or some other encoding). When shifting the sliding window, we can simply subtract the value of the character being shifted out and add that of the character being added to the sliding window.



This over-simplistic hash function will prove to be problematic if used in the Rabin-Karp algorithm since its time complexity depends on the number of collisions. One downfall of this hash function is that addition is commutative, so all permutations of a string will also have the same hash value. A better hash function is

$$H(S,i,j) = \sum_{k=i}^{j} S_k \cdot c^{j-k} = S_i \cdot c^{j-i} + S_{i+1} \cdot c^{j-i-1} + \ldots + S_j,$$

where c is a large constant. To protect against overflow, we take the hash modulo a large prime number (large so that collisions are minimized). We use the previous hash to calculate the next hash by

$$H(S, i+1, j+1) = (H(S,i,j) - S_i \cdot c^{j-i}) \cdot c + S_{j+1}.$$

Using this clever method of hashing, our overall time complexity comes out to be $O(m+n)$ on average case and $O(mn)$ in the worst case.

3.2 Knuth-Morris-Pratt Algorithm (KMP)

Another other important string matching algorithm is the Knuth-Morris-Pratt algorithm. KMP takes advantage of the way we traverse the string. When a character-character match fails, we don't have to start over with the pattern that we are searching for. Instead, we learn information about the pattern, and we use it when we find a mismatch.

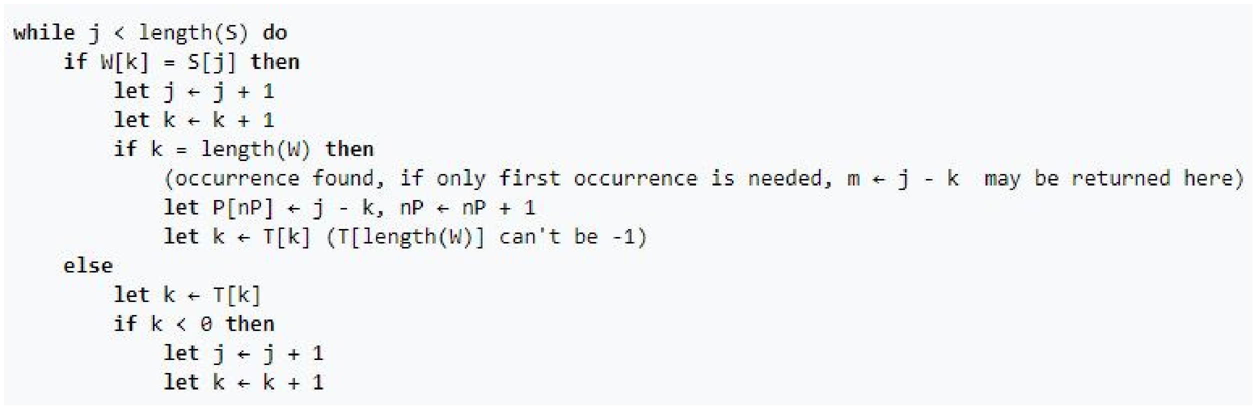
Let us call our text `S` and our pattern `P` (recall that $|\texttt{S}| = n$ and $|\texttt{P}| = m$). The first step is to iterate through the text and create a *partial match table* `T[i]`. Other than `T[0] = -1`, we set `T[i]` equal to the largest integer smaller than `i` such that `P[0...T[i]-1]` = `P[i-T[i]...i-1]` (In other words, `T[i]` holds the length of the longest prefix equal to the suffix of `P[0..i-1]`).

This partial match table is useful because it gives us information about overlaps in the pattern. When a mismatch is found, the overlapping portions provide a place to start searching in the pattern once again. Using this, we don't have to backtrack all the way to the beginning of `P`. Instead, we jump back to `T[i]`, the index of the previous overlap.

Here is how KMP works: We iterate through both `P` and `S` with pointers `i` and `j`, respectively. At each step we consider two cases, either `P[i]` = `S[j]` or `P[i]` ≠ `S[j]`. If `P[i]` = `S[j]`, then increment both `i` and `j`. Otherwise, set `i = T[i]`, unless `T[i] = -1`, in which case simply increment `i`.
Here is an example with `P` = `ababc` and `S` = `cabababcaa`:

	`j`	`S[j]`	`i`	`P[i]`	`T[i]`	
0	0	c	0	a	-1	no match
1	1	a	**0**	a	-1	
2	2	b	1	b	0	
3	3	a	2	a	0	
4	4	b	3	b	1	
5	5	a	4	c	**2**	backtrack to index 2
6	5	a	**2**	a	0	
7	6	b	3	b	1	
8	7	c	4	c	2	FINISHED

Here is pseudocode for KMP (note that the image source uses different variable names: `W` is the pattern and `P` stores the positions of the matches):



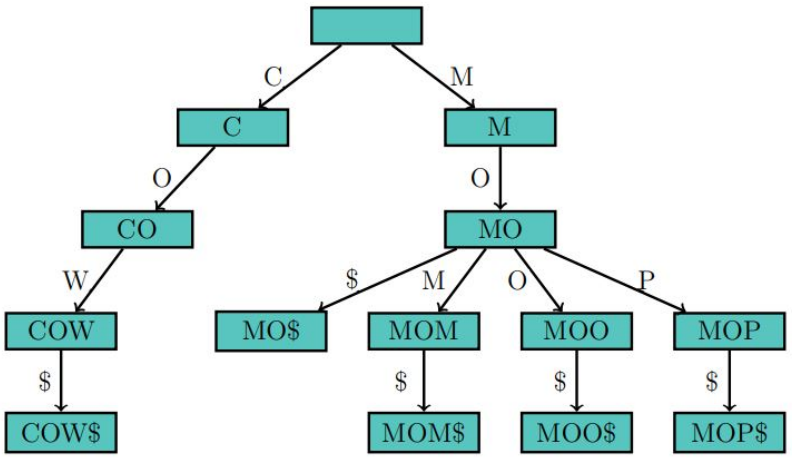
```
while j < length(S) do
    if W[k] = S[j] then
        let j ← j + 1
        let k ← k + 1
        if k = length(W) then
            (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here)
            let P[nP] ← j - k, nP ← nP + 1
            let k ← T[k] (T[length(W)] can't be -1)
    else
        let k ← T[k]
        if k < 0 then
            let j ← j + 1
            let k ← k + 1
```

The Knuth-Morris-Pratt algorithm has a worst case time complexity of $O(m + n)$: $O(m)$ for precomputation (filling up `T`) and $O(n)$ for the search.

4 String Data Structures

4.1 Trie

A trie (from reTRIEval) is a data structure for storing strings that supports insertion and look-up in linear time. The trie maintains the strings in a rooted tree, where each vertex represents a prefix and each edge is labeled with a character. The prefix of a node n is the string of characters on the path from the root to n. (In particular, the prefix of the root is the empty string.) Every string that is stored in the tree is represented by a path starting from the root. Below is a picture of a trie storing “COW,” “MO,” “MOM,” “MOO,” and “MOP.” (In order to identify the end of a string, we can append a “\$” to every string).



The advantage of the trie comes from its tree structure. Having common prefixes bundled together on the same path means we can compute compute information relating to prefixes easily.

4.2 Suffix Array

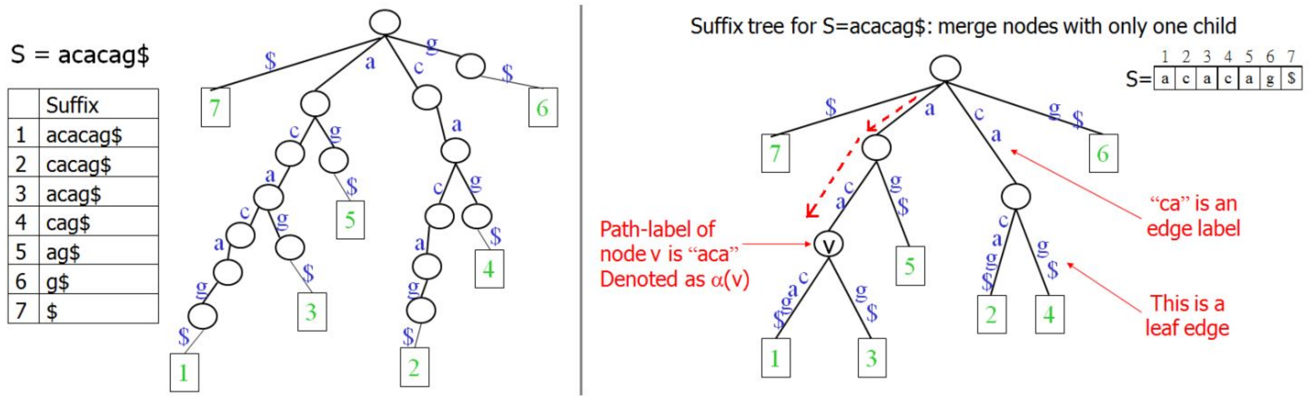
For a string of length n, each index i ($0 \leq i < n$) can represent the suffix that starts at that index. A suffix array is a list of these indices, sorted in lexicographically increasing order by suffix. In other words, a suffix array is essentially a sorted list of the suffixes of a string. Below is the suffix array of “mississippi\$.”

Why is a suffix array useful? The crucial observation is that every substring of a string is a prefix of a suffix. Thus, if we have something that does well with prefixes, such as hashing or a trie, we use this to compute information about substrings. A trie built from suffixes (with another modification) is known as a **suffix tree**, which we’ll cover next.

Suffix Array	Suffix
11	$
10	i$
7	ippi$
4	issippi$
1	ississippi$
0	mississippi$
9	pi$
8	ppi$
6	sippi$
3	sissippi$
5	ssippi$
2	ssissippi$

4.3 Suffix Tree

A suffix tree is a trie of all the suffixes of a string, along with a modification. In particular, we modify a suffix trie by merging vertices with only one child. The difference is best shown in the figure below, in which the left depicts a trie and the right shows a suffix tree.


S = acacag$
Suffix tree for S=acacag$: merge nodes with only one child
Path-label of node v is "aca" Denoted as α(v)
"ca" is an edge label
This is a leaf edge

	Suffix
1	acacag$
2	cacag$
3	acag$
4	cag$
5	ag$
6	g$
7	$

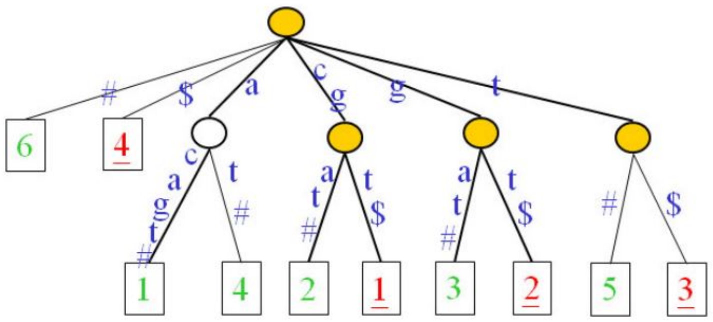
Building efficient suffix trees in a contest environment is a bit complex and risky. However, assuming you have already built a suffix tree for a string, there are a lot of useful applications. We'll see one of the applications by going back to a problem from earlier in the lecture.

4.3.1 Example: Longest Common Substring

Let's go back to the longest common substring problem. When we had previously solved this problem using dynamic programming, we had found an $O(mn)$ solution. However, using suffix trees, we can easily develop an $O(m + n)$ algorithm.

Let the two strings we are considering be $S1$ and $S2$. We can build a *generalized suffix tree* (i.e. a suffix tree for more than one string) for $S1$ and $S2$ with two different ending markers (e.g. $S1$ with # and $S2$

with \$). Next, we mark every internal vertex that has leaves which represent suffixes for *both* $S1$ and $S2$. Finally, we report the deepest marked vertex as the answer. For example, with $S1$ = "acgat#" and $S2$ = "cgt\$", the Longest Common Substring is "cg" of length 2. In the figure below, the generalized suffix tree for both strings is shown. The deepest marked vertex (shown as yellow) is "cg", which is a prefix of both "cgat#" and "cgt\$".



If two strings are of size m and n, then generalized suffix tree construction takes $O(m+n)$ and LCS finding is a DFS on tree which is again $O(m+n)$.

5 Implementations

- Java: https://github.com/indy256/codelibrary/tree/master/java/strings
- C++: https://github.com/indy256/codelibrary/tree/master/cpp/strings

6 Credits

- Competitve Programming 3
- Crash Course Coding by Samuel Hsiang
- SCT 2014 String Matching Lecture
- Stable Sort YouTube
- Stanford CS 97SI
- Wikipedia