CPSC 490 – Problem Solving in Computer Science

Lecture 11: Suffix Array

Jason Chiu and Raunak Kumar

2017/01/30

University of British Columbia

LCS or LCS?

We have solved the Longest Common Subsequence problem with DP.

What about Longest Common Substring??

Longest Common Substring

Very slow method: run Aho Corasick to find all substrings of S_1 that appear in $S_2 \rightarrow$ at least $O(m^2 + n)$

Longest Common Substring

Very slow method: run Aho Corasick to find all substrings of S_1 that appear in $S_2 \rightarrow$ at least $O(m^2 + n)$

Observation: only need to match suffixes of S_1 because Aho Corasick can tell you longest prefix match of any suffix, which is good enough.

Longest Common Substring

Very slow method: run Aho Corasick to find all substrings of S_1 that appear in $S_2 \rightarrow$ at least $O(m^2 + n)$

Observation: only need to match suffixes of S_1 because Aho Corasick can tell you longest prefix match of any suffix, which is good enough.

$\Rightarrow$ All we need to do is to figure out how to build a Suffix Trie of S_1 with all the extra arrows for Aho Corasick, and then run S_2 through.

Longest Common Substring

Very slow method: run Aho Corasick to find all substrings of S_1 that appear in $S_2 \rightarrow$ at least $O(m^2 + n)$

Observation: only need to match suffixes of S_1 because Aho Corasick can tell you longest prefix match of any suffix, which is good enough.

$\Rightarrow$ All we need to do is to figure out how to build a Suffix Trie of S_1 with all the extra arrows for Aho Corasick, and then run S_2 through.

Suffix Trie can be built in $O(n)$ but algorithm is quite complicated. Instead we will build a simpler data structure – a Suffix Array

Longest Common Substring with Suffix Array

Main idea: if we have a sorted list of all suffixes of both S_1 and S_2, then we can just scan through the list and compare adjacent suffixes.

What we will do:

- Construct a Suffix Array of a string in $O(n \log^2 n)$
  - $O(n \log n)$ if you use radix sort
  - $O(n)$ algorithms exist, but are more complicated
- At the same time, construct a DP table so that we can find Longest Common Prefix of any two suffixes in $O(\log n)$

Suffix Array Construction

To avoid $O(n^2)$ memory, store suffixes by their starting index.

Full comparison of 2 suffixes is slow – possibly O(string length), but comparing only first character is fast!

$\Rightarrow$ Let's try sorting suffixes by first character

Suffix Array Construction – Pass 1

First pass: sort by the first character of the suffix and label with rank

```
{B} 0 = BANANA$          {$} -> {0} 6 = $
{A} 1 = ANANA$           {A} -> {1} 1 = ANANA$
{N} 2 = NANA$            {A} -> {1} 3 = ANA$
{A} 3 = ANA$       =>    {A} -> {1} 5 = A$
{N} 4 = NA$              {B} -> {2} 0 = BANANA$
{A} 5 = A$               {N} -> {3} 2 = NANA$
{$} 6 = $                {N} -> {3} 4 = NA$
```

Now we know the rank of all suffixes by their first character.

Suffix Array Construction – Pass 2

Observation: a suffix of a suffix is a suffix, so if two suffixes share first character, we know their relative rank by second character!

⇒ Sort again with pair(rank 1, rank 2)

```
{0, 0} 6 = $                  {0, 0} -> {0} 6 = $
{1, 3} 1 = ANANA$             {1, 0} -> {1} 5 = A$
{1, 3} 3 = ANA$               {1, 3} -> {2} 1 = ANANA$
{1, 0} 5 = A$          =>     {1, 3} -> {2} 3 = ANA$
{2, 1} 0 = BANANA$            {2, 1} -> {3} 0 = BANANA$
{3, 1} 2 = NANA$              {3, 1} -> {4} 2 = NANA$
{3, 1} 4 = NA$                {3, 1} -> {4} 4 = NA$
```

Now we know the rank of all suffixes by their first 2 characters.

Suffix Array Construction – Pass 3

We know the rank by first 2 characters, so if two suffixes have same rank, we know their relative rank by the next 2 characters.

$\Rightarrow$ Sort again with pair(rank of char 1-2, rank of char 3-4)

```
{0, 0} 6 = $                  {0, 0} -> {0} 6 = $
{1, 0} 5 = A$                 {1, 0} -> {1} 5 = A$
{2, 2} 1 = ANANA$             {2, 1} -> {2} 3 = ANA$
{2, 1} 3 = ANA$         =>    {2, 2} -> {3} 1 = ANANA$
{3, 4} 0 = BANANA$            {3, 4} -> {4} 0 = BANANA$
{4, 4} 2 = NANA$              {4, 0} -> {5} 4 = NA$
{4, 0} 4 = NA$                {4, 4} -> {6} 2 = NANA$
```

Now we know the rank of all suffixes by first their 4 characters.
For “BANANA” we are done as all ranks are unique.

Otherwise, sort again with pair(rank of char 1-4, rank of char 5-8), etc.

Suffix Array Construction: Summary

Define rank[k][i] = the rank of S[i..n] when sorted by first 2^k chars, then the previous construction is equivalent to this DP recurrence:

- rank[0][i] = S[i] for $0 \leq i < n$
- rank[k][i] = -1 for all $i \geq n$
- rank[k][i] = rank of s[i..n] after sorting all suffixes by
  {rank[k-1][i], rank[k-1][i+2^k]}

Our suffix array is then rank[log n], but useful to keep entire array.

Time complexity: We did $O(\log n)$ sorts so $O(n \log^2 n)$

Suffix Array Construction

```
1  MAKE_SUFFIX_ARRAY(S of length N > 1):
2    initialize array R[1 + log N][N], T[N]
3    for i = 0 to N-1:
4      R[0][i] = S[i]
5    initialize skip = 1, lvl = 1
6    while skip < N:
7      for i = 0 to N-1:
8        T[i] = {{R[lvl-1][i], R[lvl-1][i+skip]}, i}
9      sort T
10     for i = 0 to N-1:
11       if i > 0 && T[i]._1 == T[i-1]._1:
12         R[lvl][T[i]._2] = R[lvl][T[i-1]._2]
13       else:
14         R[lvl][T[i]._2] = i
15     skip = skip * 2, lvl = lvl + 1
16   return R
```

Longest Common Prefix of Two Suffixes

Notice that we can use the rank array to compute LCP of two suffixes

- If rank[k][i] == rank[k][j] then LCP of S[i..n] and S[j..n] $\geq 2^k$
- $\Rightarrow$ Find largest k such that rank[k][i] = rank[k][j],
  return 2^k + LCP of S[i+2^k..n] and S[j+2^k..n]

Time complexity: similar to LCA, i.e. $O(\log n)$ time

Longest Common Prefix of Two Suffixes

```
1  LCP(i, j):
2    if i == j:
3      return N - i
4    initialize len = 0
5    for k = log N to 0:
6      if i >= N || j >= N:
7        break
8      if R[k][i] == R[k][j]:
9        len = len + 2^k
10       i = i + 2^k, j = j + 2^k
11   return len
```

Longest Common Substring – Solution

- Build suffix array of $S_1 S_2$
- Scan through suffix array, when suffixes of adjacent rank start in different string, compute LCP of them
- Make sure to clamp the LCP to boundary of the 2 strings
- Take the max of all LCPs you computed

Problem 1

Find the lexicographically least rotation of a string.

In other words, if you rotate the string in all possible ways and sort them alphabetically, what is the first string you get?

Example: BANANA $\Rightarrow$ ABANAN

Problem 1 – Solution

- Construct suffix array of two copies of S concatenated
- Output the first suffix of at least length $|S|$

Problem 2

Find the longest palindromic substring of a string

Problem 2 – Solution

Idea: palindrome is made of two pieces where the reverse of first piece shares common prefix with second piece

- Construct suffix array of S + reverse(S)
- Try all possible centers *i*: compute LCP of reverse(S[0..i]) and S[i..n-1] (or start at i+1 for even palindromes)

Note: Manacher's algorithm solves this in $O(n)$ but why think when you can just copy paste Suffix Array!

Problem 3

Count the number of distinct substrings of a string.

Hint: consider how a suffix array represents a suffix trie

Problem 3 – Solution

Reduction: count number of nodes in suffix trie

Key idea: suffix array = in order traversal of suffix trie

- Construct suffix array
- Output suffix[0].length() + $\sum_{i=1}^{n-1}$ suffix[i].length() - LCP(i-1, i)

End of material for Assignment 3

Next Class

Range Query