This course material is now made available for public usage.
Special acknowledgement to School of Computing, National University of Singapore
for allowing Steven to prepare and distribute these teaching materials.

CS3233 Competitive Programming

Dr. Steven Halim

Week 10 – String Processing

Outline

- Mini Contest #8 + Discussion + Break + Admins
- Covered *briefly* in class but *indirectly* examinable:
  - Basic String Processing Skills
- Skipped this semester (use this skill to solve more UVas):
  - Ad Hoc String Problems
  - String Matching (Knuth-Morris-Pratt's Algorithm)
- Today, focus on:
  - Suffix Trie/Tree/Array
- Note: DP on String has been discussed in Week 04-05

Section 6.2

BASIC STRING PROCESSING SKILLS

Basics of String Processing (01)
Data Structure

C (top)/C++ (bottom)

- C: **null-terminated character array**
  - We have to know the string length (or at least the upperbound) beforehand

```
char str[10000];
```

- C++: **string** class

```
#include <string>
using namespace std;
string str;
```

Java

- **String class**

```
String str;
```

Basics of String Processing (02)

Reading a String (**a word**)

C (top)/C++ (bottom)

```
#include <stdio.h>

scanf("%s", &str);
  // & optional


#include <iostream>
using namespace std;

cin >> str;
```

Java

```
import java.util.*;

Scanner sc = new
  Scanner(System.in);
str = sc.next();
```

Basics of String Processing (03)

Reading a **Line** of String

C (top)/C++ (bottom)

```
gets(str);
// alternative/safer version
// fgets(str, 10000, stdin);
// but you will read extra
// '\0' at the back
// PS: Mooshak prefer fgets


getline(cin, str);
```

Java

```
str = sc.nextLine();
```

Basics of String Processing (04)

Printing and Formatting String Output

C (top)/C++ (bottom)

- Preferred method ☺

```
printf("s = %s, l = %d\n",
  str, (int)strlen(str));
```

- C++ version is harder ☹

```
cout << "s = " << str <<
  ", l = " << str.length()
  << endl;
```

Java

- We can use **`System.out.print`** or **`System.out.println`**, but the best is to use C-style **`System.out.printf`**

```
System.out.printf(
  "s = %s, l = %d\n",
  str, str.length());
```

Basics of String Processing (05)

Comparing Two Strings

C (top)/C++ (bottom)

```
printf(strcmp(str, "test") ?
  "different\n" :
  "same\n" );


cout << str == "test" ?
  "same" :
  "different" << endl;
```

Java

```
System.out.println(
  str.equals("test"));
```

Basics of String Processing (06)

Combining Two Strings

C (top)/C++ (bottom)

```
strcpy(str, "hello");
strcat(str, " world");
printf("%s\n", str);
// output: "hello world"
```

```
str = "hello";
str.append(" world");
cout << str << endl;
// output: "hello world"
```

Java

```
str = "hello";
str += " world";
System.out.println(str);
// output: "hello world"
```

Basics of String Processing (07)

String Tokenizer: Splitting Str into Tokens

C (top)/C++ (bottom)

```
#include <string.h>
for (char *p=strtok(str, " ");
  p;
  p = strtok(NULL, " "))
    printf("%s\n", p);


#include <sstream>
stringstream p(str);
while (!p.eof()) {
  string token;
  p >> token;
  cout << token << endl;
}
```

Java

```
import java.util.*;

StringTokenizer st = new
  StringTokenizer(str, " ");
while (st.hasMoreTokens())
  System.out.println(
    st.nextToken());
```

Basics of String Processing (08)

String Matching: Finding a Substr in a Str

C (top)/C++ (bottom)

```
char *p=strstr(str, substr);
if (p)
  printf("%d\n", p-str-1);
```

```
int pos = str.find(substr);
if (pos != string::npos)
  cout << pos - 1 << endl;
```

Java

```
int pos=str.indexOf(substr);
if (pos != -1)
  System.out.println(pos);
```

Basics of String Processing (09)
Editing/Examining Characters of a String

Both C & C++

```
#include <ctype.h>

for (int i = 0; str[i]; i++)
  str[i] = toupper(str[i]);
  // or tolower(ch)
  // isalpha(ch),isdigit(ch)
```

Java

- Characters of a Java `String` can be accessed with `str.charAt(i)`, but Java `String` is immutable (cannot be changed)
- You may have to create new String or use Java `StringBuffer`

Basics of String Processing (10)

Sorting Characters of a String

Both C & C++

```
#include <algorithm>

// if using C-style string
sort(s, s + (int)strlen(s));

// if using C++ string class
sort(s.begin(), s.end());
```

Java

- Java `String` is immutable (cannot be changed)
- You have to break the string `toCharArray()` and then sort the character array

Basics of String Processing (11)
Sorting Array/Vector of Strings

Preferably C++

```
#include <algorithm>
#include <string>
#include <vector>

vector<string> S;
// assume that S has items
sort(S.begin(), S.end());
// Swill be sorted now
```

Java

```
Vector<String> S =
  new Vector<String>();
// assume that S has items
Collections.sort(S);
// S will be sorted now
```

List of (simple) problems solvable with basic string processing skills

Section 6.3

Just a splash and dash for this semester

(do a few programming exercises on your own)

AD HOC STRING PROBLEMS

Ad Hoc String Problems (1)

- Cipher (Encode-Encrypt/Decode-Decrypt)
  - Transform string given a coding/decoding mechanism
  - Usually, we need to follow problem description
  - Sometimes, we have to guess the pattern
  - UVa 10878 – Decode the Tape
- Frequency Counting
  - Check how many times certain characters (or words) appear in the string
  - Use efficient data structure (or hashing technique)
  - UVa 902 – Password Search

Ad Hoc String Problems (2)

- Input Parsing
  - Given a grammar (in Backus Naur Form or in other form), check if a given string is valid according to the grammar, and evaluate it if possible
  - Use **recursive** parser, Java **Pattern (RegEx)** class
  - UVa 622 – Grammar Evaluation
- Output Formatting
  - The problematic part of the problem is in formatting the output using certain rule
  - UVa 10894 – Save Hridoy

Ad Hoc String Problems (3)

- String Comparison
  - Given two strings, are they similar with some criteria?
  - Case sensitive? Compare substring only? Modified criteria?
  - UVa 11233 – Deli Deli
- Others, not one of the above
  - But still solvable with just basic string processing skills
- Note:
  - None of these are likely appear in IOI other than as the bonus problem per contest day (no longer true in 2011)
  - In ICPC, one of these can be the bonus problem

Knuth-Morris-Pratt's Algorithm

Section 6.4

Skipped this semester (please use Suffix Array for (long) String Matching)

STRING MATCHING

String Matching

- Given a pattern string P,
  can it be found in the longer string T?
  - Do not code naïve solution
  - Easiest solution: Use string library
    - C++: string.find
    - C: strstr
    - Java: String.indexOf
  - In CP2.9 book: KMP algorithm
  - Or later/after this: Suffix Array

The earlier form of this teaching material is credited to
A/P Sung Wing Kin, Ken from SoC, NUS

CP2.9 Section 6.6

SUFFIX TRIE, TREE, AND ARRAY

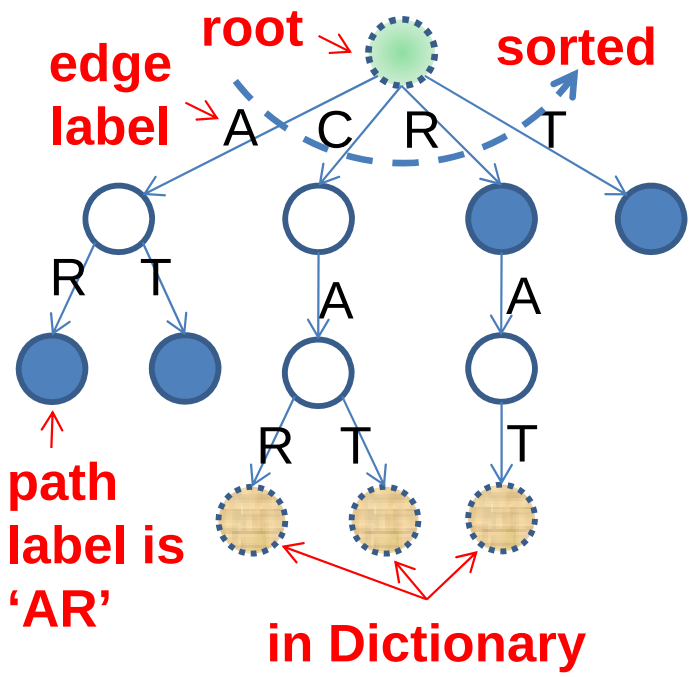
Suffix Trie (‘CAR’, ‘CAT’, ‘RAT’)

All Suffixes:

1. CAR
2. AR
3. R
4. CAT
5. AT
6. T
7. RAT
8. AT
9. T

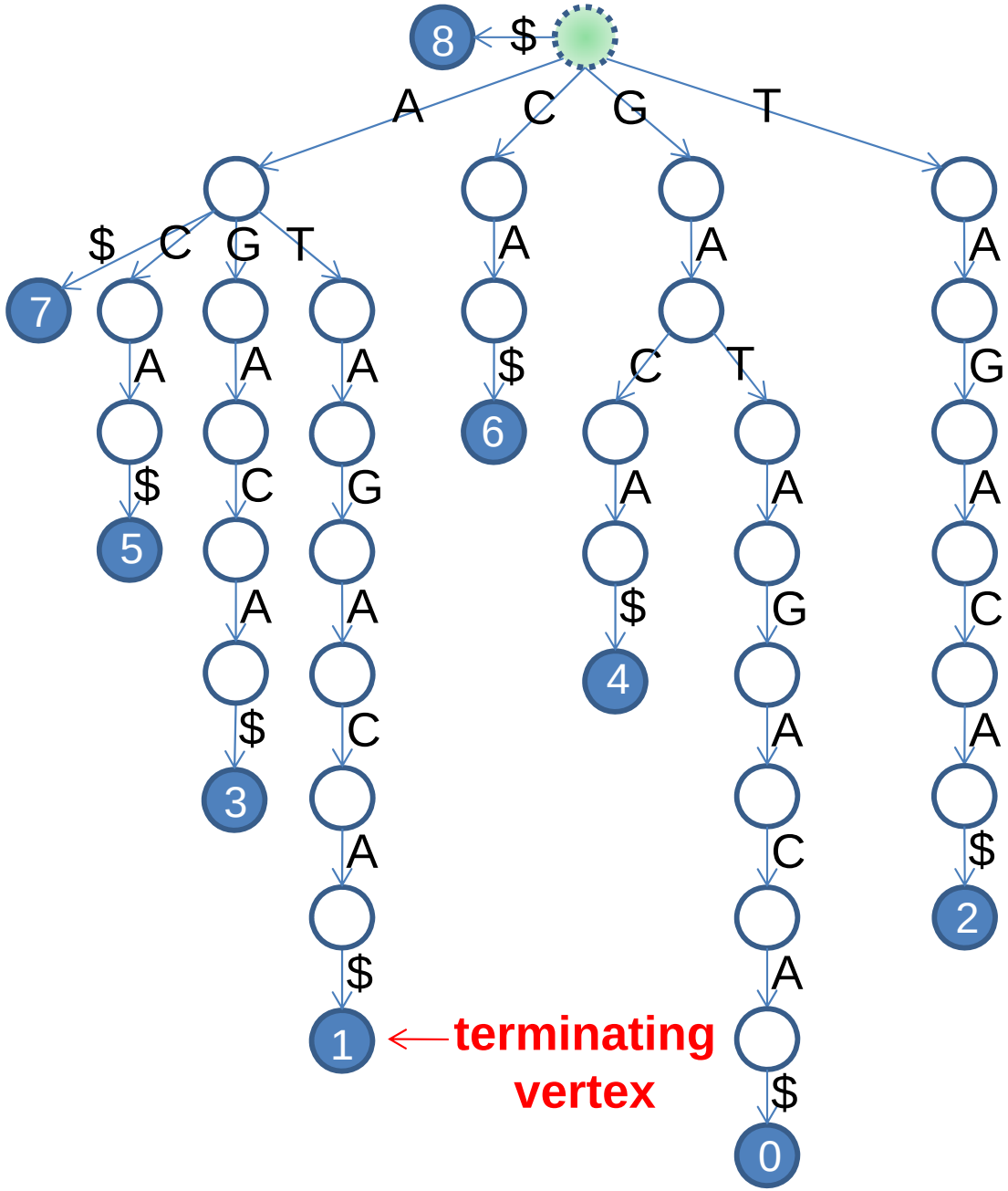
Sorted Unique Suffixes:

1. AR
2. AT
3. CAR
4. CAT
5. R
6. RAT
7. T



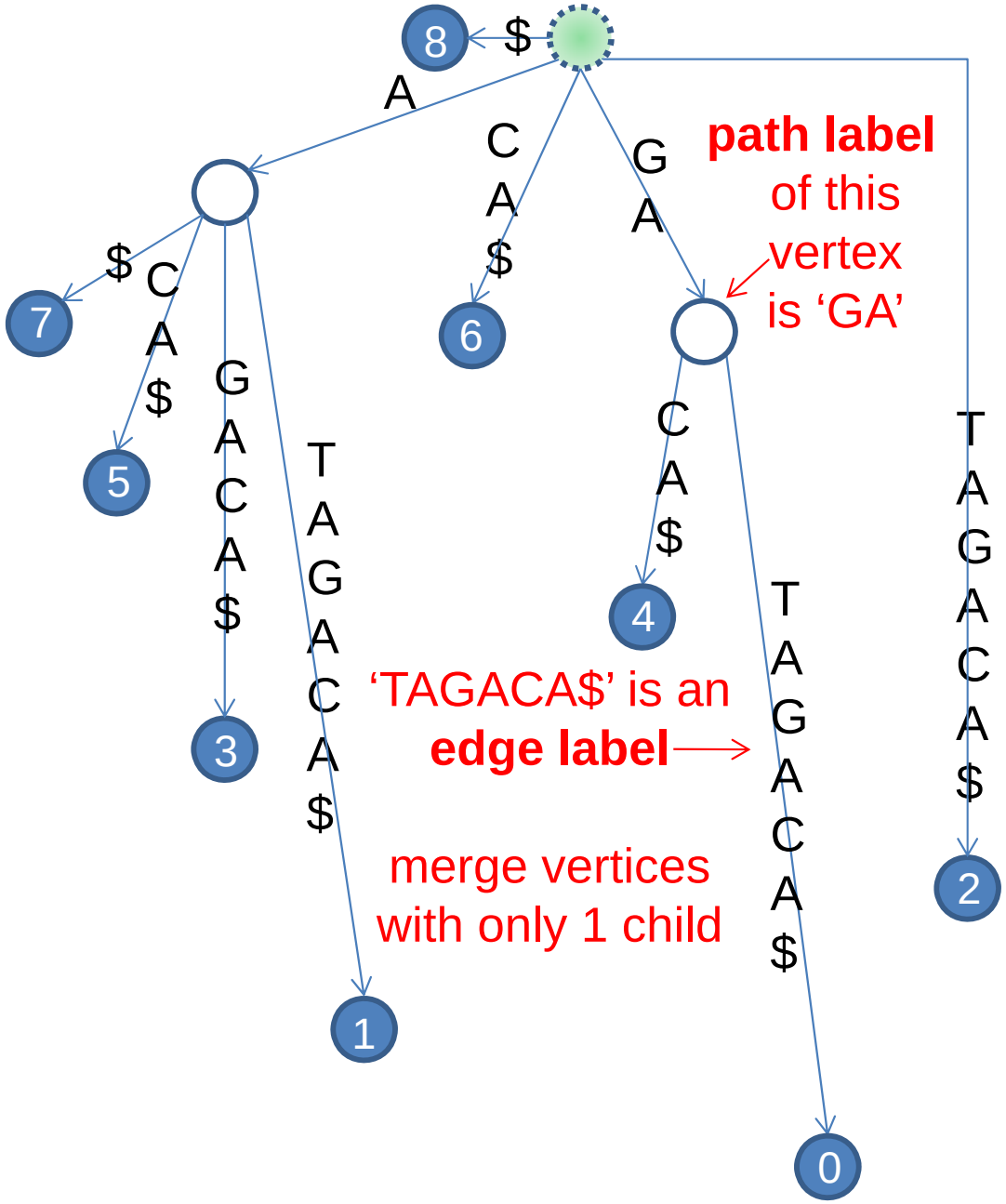
Suffix Trie (T = ‘GATAGACA$’)

i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$



Suffix Tree (T = ‘GATAGACA$’)

i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

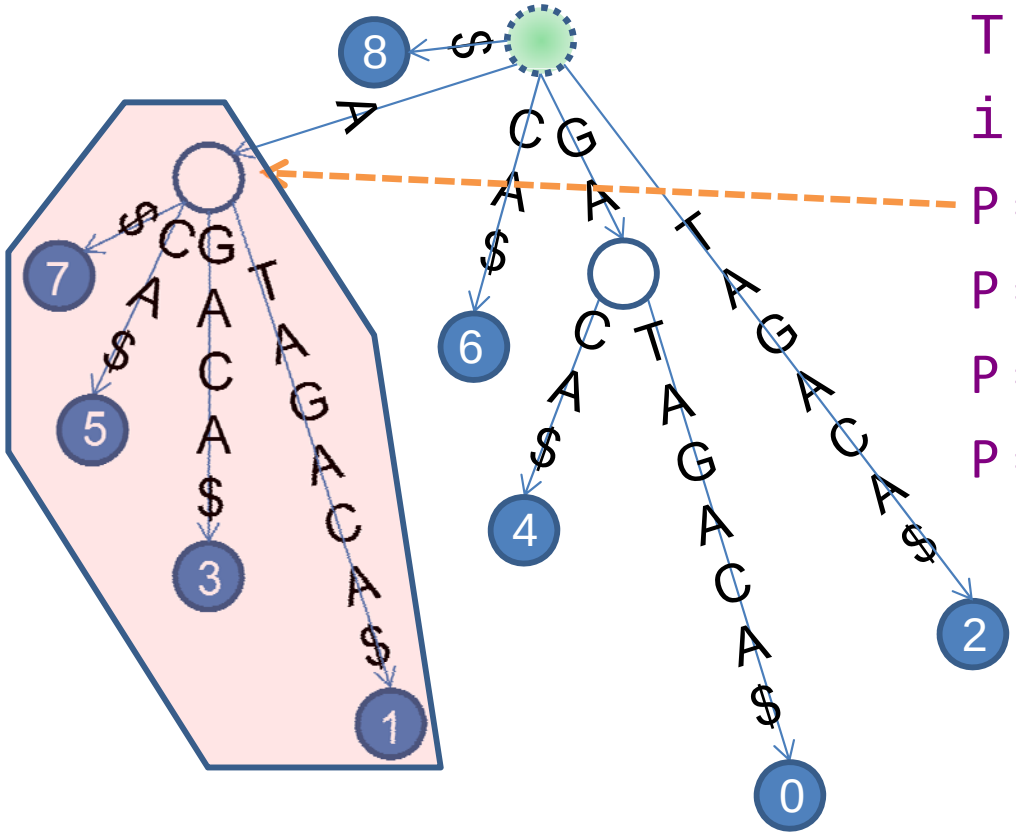


What can we do with this specialized string data structure?

APPLICATIONS OF SUFFIX TREE

String Matching

- To find all occurrences of **P** (of length m) in **T** (of length n)
  - Search for the vertex **x** in the Suffix Tree which represents **P**
  - All the leaves in the subtree rooted at x are the occurrences
- Time: O(**m** + occ) where occ is the total no. of occurrences



T = 'GATAGACA$'

i = '012345678'

P = 'A' → Occurrences: 7, 5, 3, 1

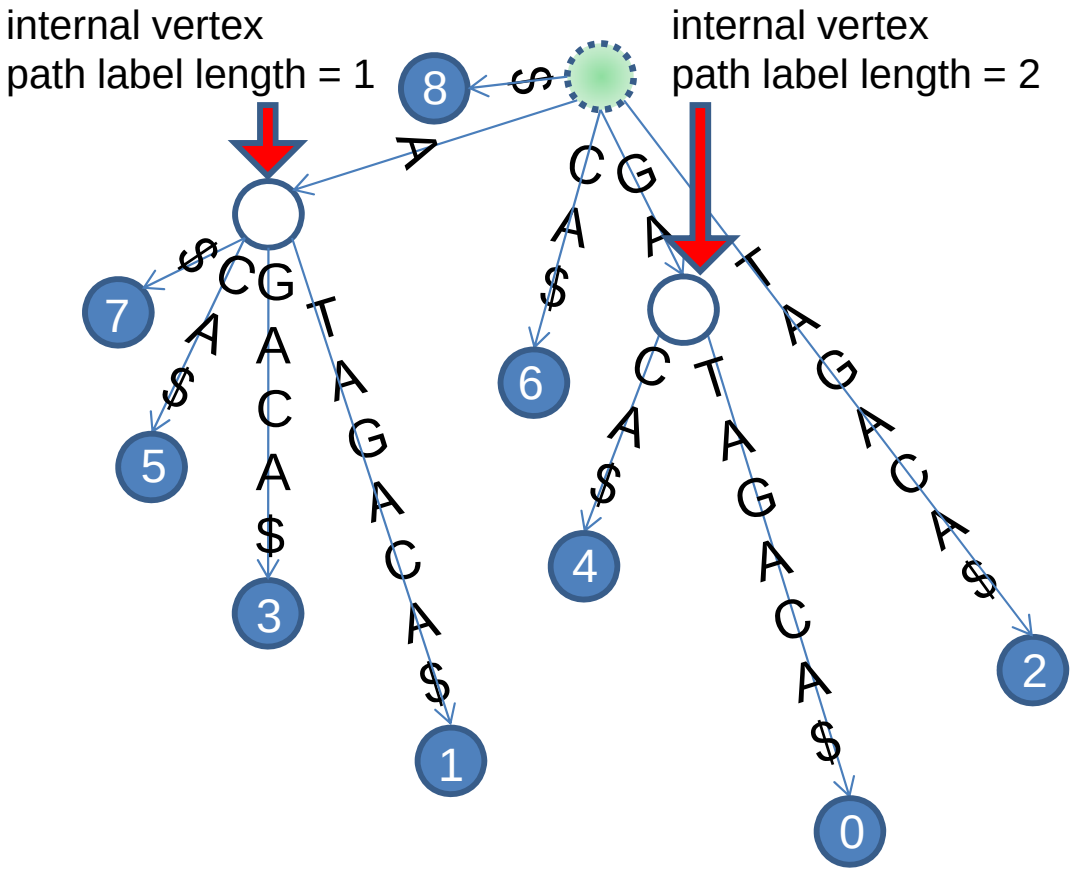
P = 'GA' → Occurrences: 4, 0

P = 'T' → Occurrences: 2

P = 'Z' → Not Found

Longest Repeated Substring

- To find the longest repeated substring in T
  - Find the deepest internal node
- Time: O(n)



e.g. T = 'GATAGACA$'
The longest repeated substring is 'GA' with path label length = 2

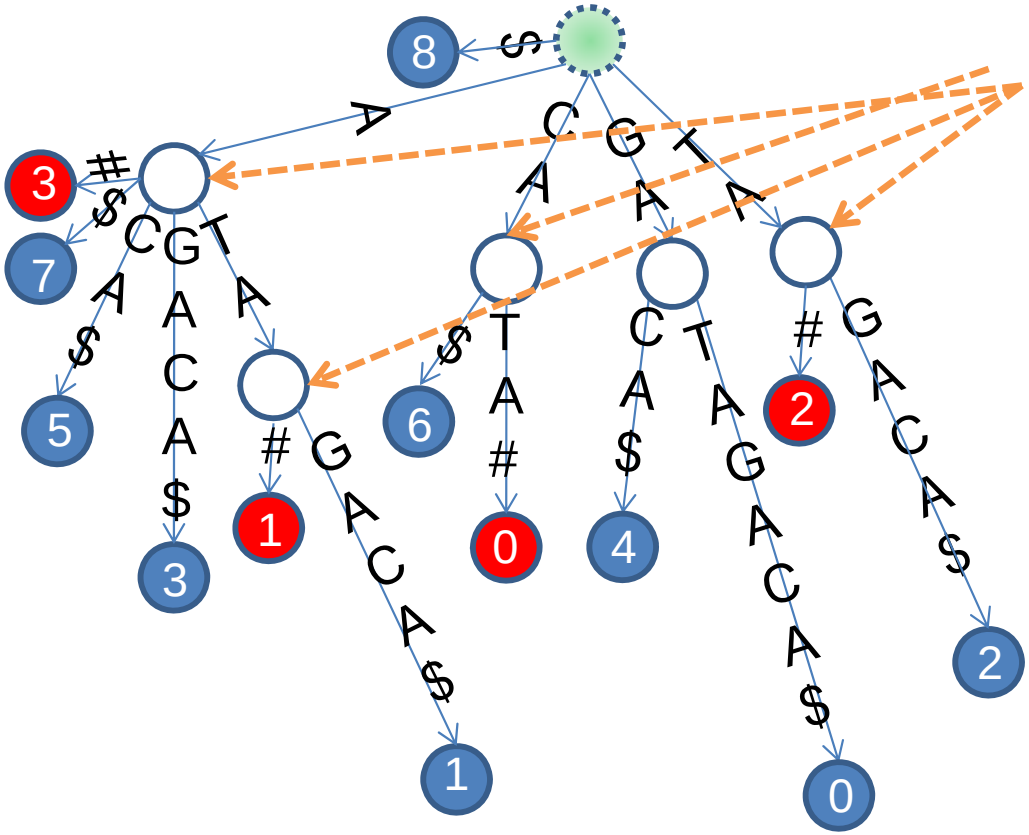
The other repeated substring is 'A', but its path label length = 1

Longest Common Substring

- To find the longest common substring of two or more strings
  - Note: In 1970, Donald Knuth conjectured that a linear time algorithm for this problem is impossible
  - Now, we know that it can be solved in linear time
  - E.g. consider two string T1 and T2,
    - Build a **generalized** Suffix Tree for T1 and T2
      - i.e. a Suffix Tree that combines both the Suffix Tree of T1 and T2
    - Mark internal vertices with leaves representing suffixes of both T1 and T2
    - Report the deepest marked vertex

Example of LC Substring

- T1 = ‘GATAGACA$’ (end vertices labeled with blue)
  T2 = ‘CATA#’ (end vertices labeled with red)
  - Their longest common substring is ‘ATA’ with length 3



How to build Suffix Tree?

For programming contests, we use Suffix Array instead…

SUFFIX ARRAY

Disadvantage of Suffix Tree

- Suffix Tree is space inefficient
  - It requires $O(n|\Sigma|\log n)$ bits
    - N nodes, each node has $|\Sigma|$ branches, each pointer needs O(log n) bits
- Actual reason for programming contests
  - It is harder to construct Suffix Tree
- Manber and Myers (SIAM J. Comp 1993) proposes a new (in 1993) data structure, called the Suffix Array, which has a similar functionality as Suffix Tree
  - Moreover, it only requires O(n log n) bits
- And it is much easier to implement

Suffix Array (1)

- Suffix Array (SA) is an array that stores:
  - A permutation of **n** indices of sorted suffixes
  - Each integer takes O(log n) bits, so SA takes O(n log n) bits
- e.g. consider T = 'GATAGACA$'

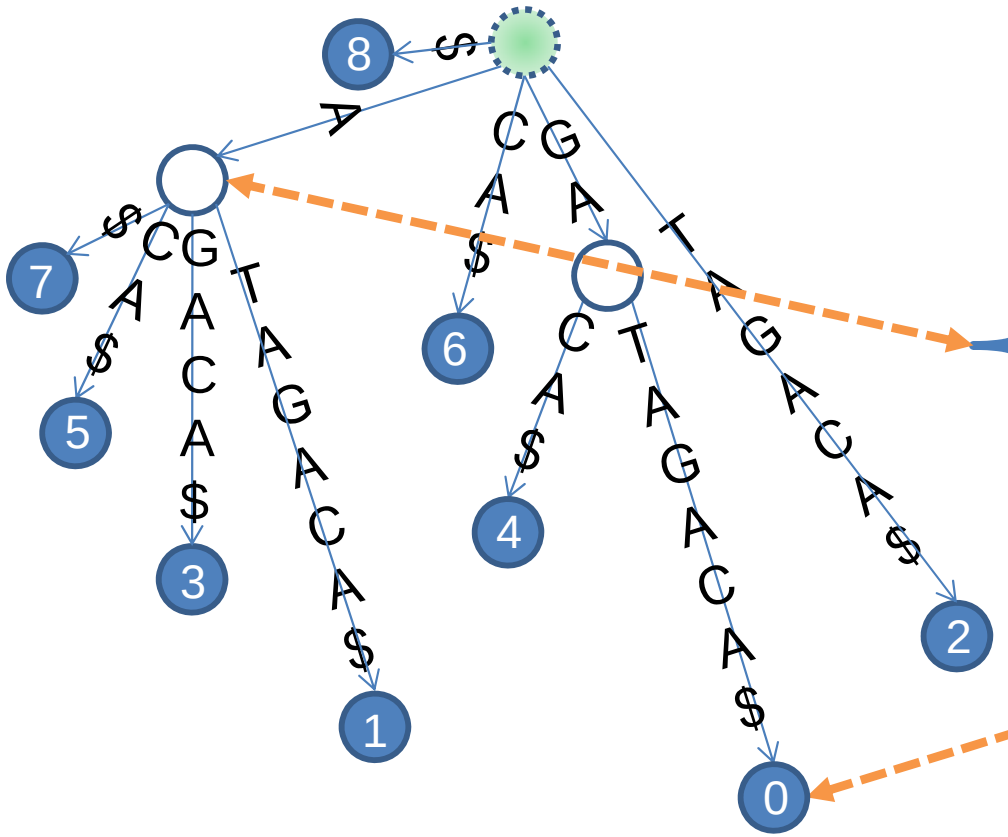
i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

Sort →

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Suffix Array (2)

- Preorder traversal of the Suffix Tree visits the terminating vertices in Suffix Array order
- **Internal vertex** in ST is a **range** in SA
  - **Each terminating vertex** in ST is an **individual index** in SA = a suffix



i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Easy/Slow Suffix Array Construction

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

char T[MAX_N]; int SA[MAX_N];

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }

int main() {
   int n = (int)strlen(gets(T));
   for (int i = 0; i < n; i++) SA[i] = i;
   sort(SA, SA + n, cmp);
}
```

This is O(N)

What is the time complexity?
Can we do better?

Overall $O(N^2 \log N)$

Most (if not all) applications related to Suffix Tree can be solved using Suffix Array

- With some increase in time complexity

APPLICATIONS OF SUFFIX ARRAY

String Matching

- Given a Suffix Array SA of the string T
- Find occurrences of the pattern string P
- Example
  - T = ‘GATAGACA$’
  - P = ‘GA’
- Solution:
  - Use Binary Search twice
    - One to get lower bound
    - One to get upper bound

String Matching Animation

Finding P = 'GA'

Finding lower bound

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Finding upper bound

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Time Analysis

- Binary search runs at most O(log n) comparisons
- Each comparison takes at most O(m) time
- We run binary search twice
- In the worst case, O(2m log n) = O(m log n)

Longest Repeated Substring

- Simply find the highest entry in LCP array
  - O(n)

i	SA[i]	LCP[i]	Suffix
0	8	0	$
1	7	0	A$
2	5	1	ACA$
3	3	1	AGACA$
4	1	1	ATAGACA$
5	6	0	CA$
6	4	0	GACA$
7	**0**	**2**	**GATAGACA$**
8	2	0	TAGACA$

Recall:
LCP = Longest Common Prefix between two successive suffices

Longest Common Substring

- T1 = 'GATAGACA$'
- T2 = 'CATA#'
- T = 'GATAGACA$CATA#'
- Find the highest number in LCP array provided that it comes from two suffices with different owner
  - Owner: Is this suffix belong to string 1 or string 2?
- O(n)

i	SA[i]	LCP[i]	Owner	Suffix
0	13	0	2	#
1	8	0	1	$CATA#
2	12	0	2	A#
3	7	1	1	A$CATA#
4	5	1	1	ACA$CATA#
5	3	1	1	AGACA$CATA#
6	10	1	2	ATA#
7	**1**	**3**	**1**	**ATAGACA$CATA#**
8	6	0	1	CA$CATA#
9	9	2	2	CATA#
10	4	0	1	GACA$CATA#
11	0	2	1	GATAGACA$CATA#
12	11	0	2	TA#
13	2	2	1	TAGACA$CATA#

Summary

- In this lecture, you have seen:
  - Various string related tricks
  - Focus on Suffix Tree and Suffix Array
- But… you need to practice using them!
  - Especially, scrutinize my Suffix Array code
  - Solve at least one UVa problem involving SA
  - We will have SA-contest next week ☺
    - 2 SA problems in A/B/C

References

- CP2.9, Chapter 6
- Introduction to Algorithms, 2nd/3rd ed, Chapter 32