Programming Challenges (GB21802)

Week 7 - String Manipulation

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: May 29, 2021)

Version 2021.1

String Problems

The manipulations of string is a common task in real life applications such as:

- Analysis of Bioinformatics Gene Data;
- Pre-processing/wrangling, of API data (ex: JSON)
- Text processing from human interfaces (natural language)

Characteristics of String Problems

- "Parsing" of inputs with special rules;
- Using Dynamic Programming for finding patterns;
- Special data structures for storing patterns;

Topics for this week

We will study the following topics this week

- String basics and ad-hoc problems
  - Basic string libraries in C++ and Java;
  - Examples of Ad-hoc problems;
- String Matchign Algorithm;
  - Naive search;
  - KMP;
  - Z-Algorithm;
- Strings algorithms with DP;
  - Edit Distance
  - Common substring
  - Palindromes
- Suffix Tree and Suffix Array;

String Basic Operations

String Representation

```
// C/C++ (ends with '\0')
char[100] str;

#include<string>
str s;
```

```
// JAVA
String str;

// JAVA strings are imutable!
// Modifying them = new object
```

Data Input

```
// Word
scanf("%s",&str); cin >> str;

// Line
gets(str);
fgets(str,1000,strdin);
getline(cin,str);
```

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

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

String Basic Operations

Testing Two Strings for Equality

```
// C/C++
result = strcmp(str,"test");
result = (str == "test");
```

```
// JAVA
result = str.equals("test");
```

Combining Two or More Strings

```
strcpy(str,"hello");
strcat(str," world");
str = "hello";
str.append(" world");
```

```
str = "hello";
str += " world";
// Careful!
// Creates new strings
```

Editing/Testing single characters in a string

```
#include <ctype.h>
for (int i=0;str[i];i++)
    str[i] = toupper(str[i])
```

```
// Java Strs are immutable
// create a new string
// or use StringBuffer
```

String Basic Operations

String Tokenizer – Separates a string based on a character

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

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

```
// JAVA
import java.util.*;
StringTokenizer st = new
  StringTokenizer(str," ");
while (st.hasMoreTokens())
  System.out.println(
     st.nextToken());
```

String Basic Operations

Finding a Substring

```
// C/C++
char *p=strstr(str,substr);
if (p) printf("%d",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);
```

Sorting Characters in a string

```
#include <algorithm>
sort(s, s+(int)strlen(s));
sort(s.begin(),s.end());
```

```
//Immutable, break the
//string using
//toCharArray()
```

Ad-hoc String Problems

Let's see some general string problems that can be solved using the string library functions that we just reviewed.

If you have difficulty in these problems, try using the **Complete Search** approach on them first!

Immediate Decodability

Problem Outline

Given a set of **2 to 8 binary words**, of length between **1 and 10**, decide if the set is **immediately decodable.**

Immediate decodable means **no word is a prefix of another word.**

Input example 1 (Decodable)

- 001
- 110
- 10101
- 01101
- 100

Input example 2 (not decodable)

- **001**
- 10101
- **00101**
- 11011
- 1011

QUIZ: How do you solve this problem?

Immediate Decodability

Hints

Input example 1 (Decodable)

- 001
- 110
- 10101
- 01101
- 100

Input example 2 (not decodable)

- **001**
- 10101
- **00101**
- 11011
- 1011

- A simple way to solve is to test every pair of strings, to see if one is a prefix of another;
  - What is the difference between prefix and substring?
  - How many steps this algorithm takes?
- You can improve this algorithm if you reduce the number of comparisons;
  - How can you prune the algorithm?
  - Does the order of the strings matter?

Ad-hoc Problem 2 – Caesar Cypher

Problem Outline

A **rotational cypher** transforms *plaintext* to *cyphertext* by adding a constant value "k" to every character.

Example: I LOVE YOU + ($k = 3$) $\rightarrow$ LCORYHCARY

Given a dictionary of plaintext, find the best translation of the cyphertext.

```
THIS  DAWN  THAT    || INPUT:  BUUBDLA PSSPABUAEBXO
ZORRO OTHER AT      || OUTPUT: ATTACK ZORRO AT DAWN
THING THE
```

QUIZ: How do we solve this problem?

Ad-hoc Problem 2 – Caesar Cypher

```
THIS  DAWN  THAT    || INPUT:  BUUBDLA PSSPABUAEBXO
ZORRO OTHER AT      || OUTPUT: ATTACK ZORRO AT DAWN
THING THE
```

- Our objective is to find the rotation that fits the largest number of words in the dictionary.
- Try every rotation, for each rotation see if the words are substrings.
- This is a very slow approach. Can it be faster?

String Matching

Definition

Given a string T (also called **text**), we want to test if the substring P (also called **pattern**) exists in T.

If P exists in T, we want to know the **index** of the start of P in T.

Example:

```
T: STEVEN EVENT
P: EVE              indexes: 2 and 7
P: EVENT            indexes: 7
P: EVENING          indexes: -1 or NULL
```

String Matching and Libraries

How do we solve string matching problems?

Use your language's string library!

- In C/C++: strstr(T,P) or T.find(P)
- In Java: T.indexOf(P)

No bugs! Usually very efficient!

But...

- Maybe you have a specific matching function (1 equals l)
- Maybe your string changes over time;
- Maybe you have to match multiple strings at the same time;
- Maybe you have to string match in a graph;
- etc...

String Matching: Complete Search

For every character T_i, test if P begins at that position.

```
for (int i = 0; i < |T|; i++)
  bool match = true;
  for (int j = 0; j < |P| && match; j++)
    if (i+j >= |T| || P[j] != T[i+j])
      match = false;
  if (match)
    printf("Match P at index %d\n", i);
```

Number of Steps:

- Average case: $O(n)$ – For natural T and small P;
- Worst case: $O(mn)$ – For programming challenges;
    - T = AAAAAAAAAAAAB
    - P = AAAAAAAB

The Knuth-Morris-Pratt (KMP) Algorithm

- The naive algorithm can be very expensive if the prefix of P happens many times in T.
- In 1977, Knuth, Morris and Pratt developed an algorithm that **uses these prefixes** to realize fast string matching.

Basic Idea

- The KMP algorithm works by identifying "borders" in the partial match between P and T.
- These borders are characterized by identical prefixes and sufixes in the T-P match.
- The algorithm uses these matches to advance the indexes of T and P, greatly reducing the number of comparisons.

The KMP algorithm is O(P+T).

KMP Algorithm – Simulation

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
// for i from 0 to 13, KMP works like full search

                  SEVENTY SEVEN
// Here, the collision is at i=25, j = 11, But because "SEV" is
// a "border", i stays the same and j is rewinded to 3

                                  SEVENTY SEVEN
// Here we find a match with i=43, j=13; SEVEN is a border, so j
// is rewinded to 5, and i is kept the same. The algorithm
// continues matching at i=44, j=5 ("T")

                                          SEVENTY SEVEN
// KMP finds a second match
```

KMP Algorithm – Rewind Array

To avoid repeated matches, the KMP algorithm builds a **rewind table** *b* (back).

```
      0  1  2  3  4  5  6  7  8  9  0  1  2  3
P =   S  E  V  E  N  T  Y     S  E  V  E  N  \0
b =  -1  0  0  0  0  0  0  0  0  1  2  3  4  5
```

Following the table *b*, we know that if we find a mismatch at $j = 11$, then we need to rewing j to $b[11] = 3$ to continue matching.

The text index i, on the other hand, will stay the same, and go forward by 1 if $b[j] = -1$.

KMP Algorithm – PseudoCode

```
char T[MAX_N], P[MAX_N];
int b[MAX_N], n, m;

void kmpPreprocess() {      // Create the Back Array
  int i = 0, j = -1; b[0] = -1;
  while (i < m) {
     while (j >= 0 && P[i] != P[j]) j = b[j];
     i++; j++;
     b[i] = j; }}

void kmpSearch() {          // Search the substring
  int i = 0, j = 0;
  while (i < n) {
     while (j >= 0 && T[i] != P[j]) j = b[j];
     i++; j++;
     if (j == m) {
        printf("P is found at index %d in T\n", i - j);
        j = b[j]; }}}
```

String Matching with the Z-Algorithm

Another linear algorithm that can perform string matching is the **Z algorithm**.

The Z algorithm constructs a **Z array**. For every index $i \in S$, $Z[i]$ is the size of the prefix of S that begins in i.

```
T = AASABAABAAT, P = AAB, S = P$T

... Build Z Array ...
S    = AAB$AASABAABAAT
Z[S]= X10021010310210
                 ^
                 String matched here. Z[i] = Len(P)
```

Z-Algorithm – Pseudocode

```
void Zarray(string S, int Z[]) {
    int n = S.length(); int L, R, k;
    L = R = 0;                // Prefix counters
    for (int i = 1; i < n; i++) {
        if (i > R) {          // Full search of prefix
            L = R = i;
            while (R < n && S[R] == S[R-L]) R++;
            Z[i] = R-L; R--;
        } else {              // Inside prefix candidate
            k = i-L;
            if (Z[k] < R-i+1) Z[i] = Z[k]; // no extension
            else {                         // prefix extension
                L = i;
                while (R < n && S[R] == S[R-L]) R++;
                Z[i] = R-L; R--;
}   }   }   }
```

Simulation: `https://personal.utdallas.edu/~besp/demo/John2010/z-algorithm.htm`

Z algorithm or KMP algorithm?

Should you use the Z algorithm or the KMP algorithm?

- Both algorithms have the same time complexity: $O(T + P)$
- Which algorithm is easier to understand?
  - KMP calculates a recursive suffix state machine for P;
  - Z-algorithm calculates a substring size array for T;

String Algorithms with Dynamic Programming

Some string problems can be described as a **search problem.** In this section, we will introduce two common problems in programming challenges that can be solved with DP algorithms:

- String Alignment/Edit Distance
- Longest Common Subsequence

It is interesting to note that substring matching is also a search problem, and that KMP / Z-algorithms can be seen as a kind of memoization.

String DP: String Alignment

The **String Alignment**[1] problem is defined as follows. Align two strings, A and B, with the maximum "alignment score":

- Character A[i] and B[i] match: do nothing, score +2
- Character A[i] and B[i] mismatch: replace A[i], score -1
- Insert a space in A[i]: score -1
- Delete A[i] (equals to insert in B[i]): score -1

```
               non-optimal    optimal
A: ACAATCC   | A_CAATCC     | A_CAATCC
B: AGCATGC   | AGCATGC_     | AGCA_TGC
score:       | 2-22--2- = 4 | 2-22-2-2 = 7
```

[1] Also called Edit Distance or Levenhstein Distance, used by spellchecking algorithms!

String Alignment: Bottom Up DP

```
                    non-optimal    optimal
A: ACAATCC      | A_CAATCC      | A_CAATCC
B: AGCATGC      | AGCATGC_      | AGCA_TGC
score:          | 2-22--2- = 4  | 2-22-2-2 = 7
```

The **Full search** approach requires recursively testing each of three options for each A[i] ($O(3^n)$). We can solve this in $O(n^2)$ using DP:

- $V(i, j)$: optimal score for prefix $A[1..i], B[1..j]$
- Start condition:
  - $V(0, 0) = 0$ (Do nothing)
  - $V(i, 0) = -1 \times i$, $V(0, j) = -1 \times j$ (delete A or B)
- Recurrence: $V(i, j) = \max(C_1, C_2, C_3)$, where
  - $C_1 = V(i-1, j-1) + \text{score}(A[i], B[j])$ // Score of match/mismatch;
  - $C_2 = V(i-1, j) + \text{score}(A[i], \_)$ // Delete $A[i]$;
  - $C_3 = V(i, j-1) + \text{score}(\_, B[j])$ // Delete $B[j]$;

String Alignment: Bottom Up DP

Simulation Matching AGCATGC and ACAATCC

- Recurrence: $V(i,j) = \max(C_1, C_2, C_3)$, where
  - $C_1 = V(i-1, j-1) + \text{score}(A[i], B[j])$ // Score of match/mismatch;
  - $C_2 = V(i-1, j) + \text{score}(A[i], \_)$ // Delete $A[i]$;
  - $C_3 = V(i, j-1) + \text{score}(\_, B[j])$ // Delete $B[j]$;

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1							
C	-2							
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

Longest Common Subsequence in Strings

Problem Definition

Given strings *A* and *B*, what is their longest common subsequence?

```
A   :    'ACAATCC'       - A_CAAT_CC
B   :    'AGCATGC'       - AGCA_TGC_
LCS:      AC AT C        - A_CA_T_C_ : ACATC
```

- We can solve LCS using a modification of String Aligment;
- Use String Alignment DP, but change costs:
  - Cost of Mismatch: $-\infty$
  - Cost of insert/deletion: 0
  - Cost of Matching: 1

Longest Palindrome

Problem Description

A **palindrome** is a string S where $S = \text{rev}(S)$. For example: MADAM.

Given a string T, what is the **longest palindrome** that you can create by deleting characters from T?

Examples:

- ADAM – ADA
- MADAM – MADAM
- NEVERODDOREVENING – NEVERODDOREVEN
- RACEF1CARFAST – RACECAR

QUIZ: Can you solve with Full Search? String Alignment DP? Others?

Longest Palindrome

Problem Description

Given a string S of size up to $N = 1000$ characters, what is the longest palindrome that you can make by deleting characters from S?

DP Solution:

- State Table:
  - len(i,j) - The largest palindrome found between i and j
- Start Conditions:
  - If $l = r$ then $\text{len}(l, r) = 1$.
  - If $r = l + 1$ and $S[l] = S[r]$, $\text{len}(l, r) = 2$, else $\text{len}(l, r) = 1$.
- Transition:
  - If $S[l] = S[r]$, then $\text{len}(l, r) = 2 + \text{len}(l + 1, r - 1)$;
  - else $\text{len}(l, r) = \max(\text{len}(l + 1, r), \text{len}(l, r - 1))$

This DP has complexity $O(n^2)$

Longest Palindrome

Longest Palindrome DP: Diagonal Table Top Down

```
       len(l,r)               len(l,r)         transition:
      final state           initial state   - A[l] == A[r]?
                                              len(l+1,r-1)+2
  R A C E F 1 C A R     R A C E F 1 C A R   - A[1] != A[r]?
                                              max(left,down)
R 1 1 1 1 1 1 3 5 7   R 1 1
A   1 1 1 1 1 3 5 5   A   1 1
C     1 1 1 1 3 3 3   C     1 1
E       1 1 1 1 1 1   E       1 1
F         1 1 1 1 1   F         1 1
1           1 1 1 1   1           1 1
C             1 1 1   C             1 1
A               1 1   A               1 1
R                 1   R                 1
```

Suffix Trie: Definition

Definition

Data structure used to find matching suffixes of multiple strings.

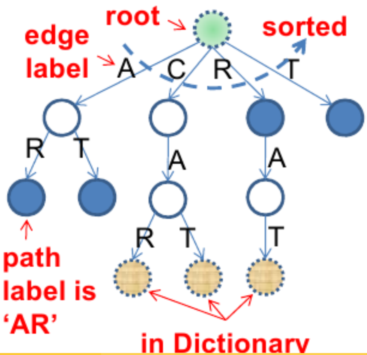
Suffix Trie for {'CAR','CAT','RAT'}

All Suffixes

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

Sorted, Unique Suffixes

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



Suffix Trie: Using it for a single, long string

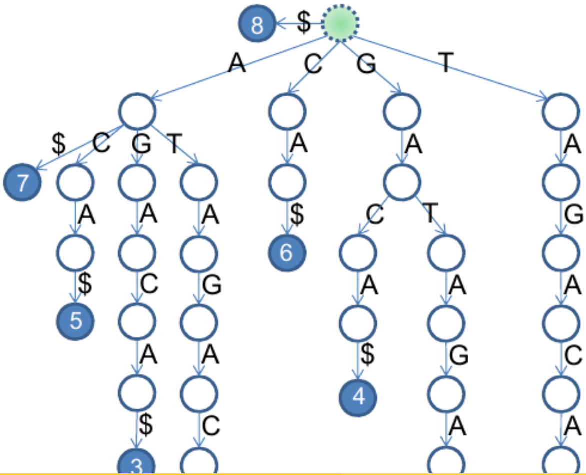
Create all n suffixes:

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

Count the occurence of substring m:

- 'A': 4 times
- 'GA': 2 times
- 'AA': 0 times

Suffix Trie (T='GATAGACA$')

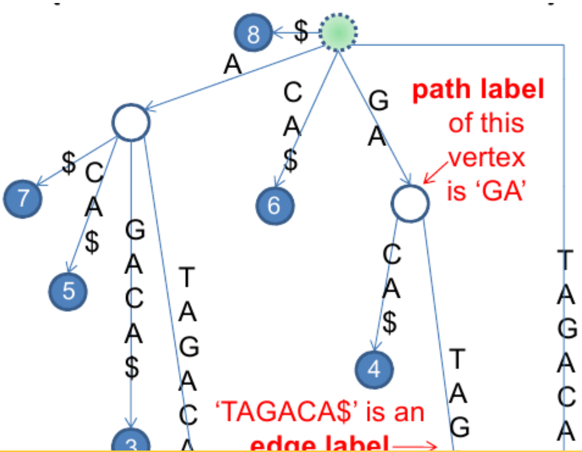


Suffix Trie: Suffix Tree

Suffix Trie (T='GATAGACA$')

Compress single child nodes to obtain "Suffix Tree"

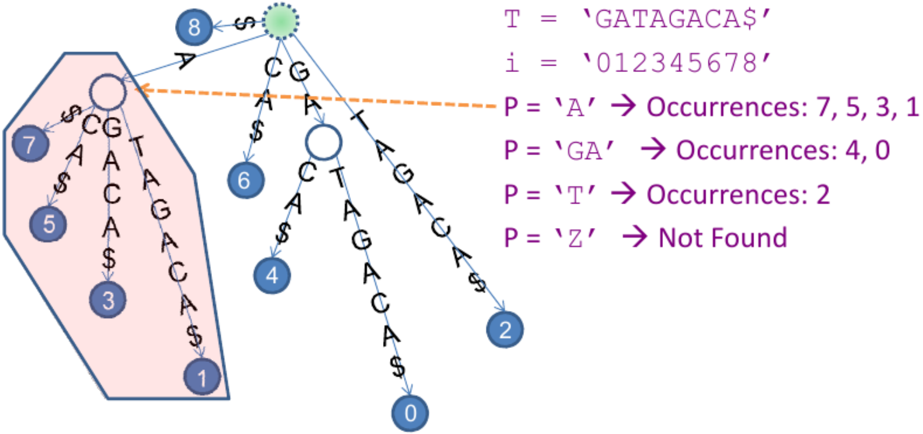
i	suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$



With the suffix tree, many algorithms become faster.

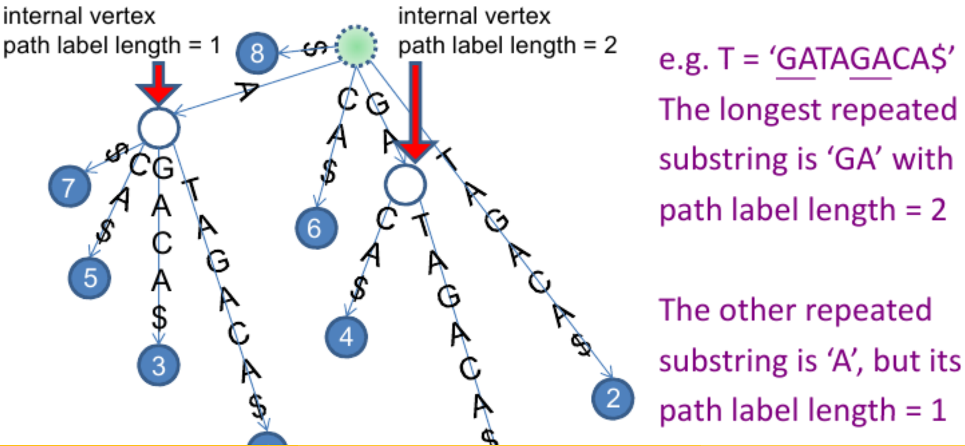
Uses of a Suffix Tree 1: String Matching

Assuming that we have the Suffix Tree already built, we can find all occurrences of substring m in T in time $O(m + \text{occ})$, where occ is the number of occurrences.



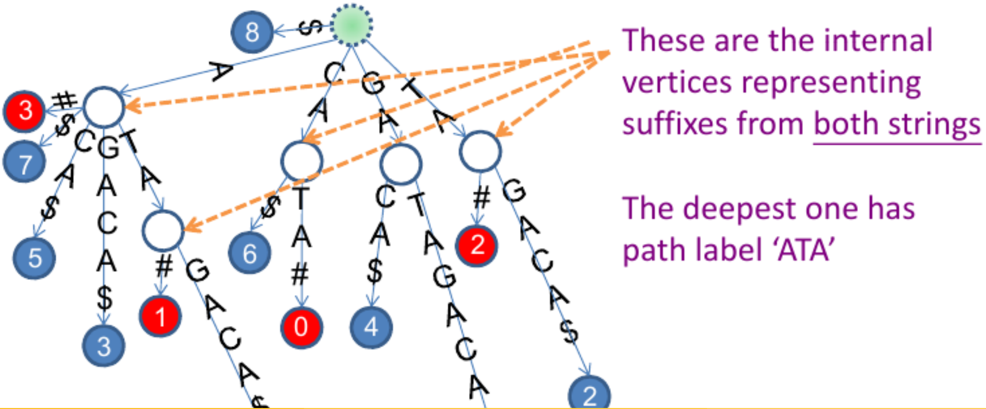
Uses of a Suffix Tree 2: Longest Repeated Substring

- The LRS is the longest substring with number of occurrences > 2;
- The LRS is the deepest internal node in the tree;



Uses of a Suffix Tree 3: Longest Common Substring

- We can find the common substring of *M* and *N* by making a combined Suffix Tree. Each string has a different ending character.
- The common substring is the deepest node that has both characters.



Suffix Trie: Suffix Array (1)

- The algorithms in previous slides are very efficient...
  ... if you have the suffix tree
- The suffix tree can be built in $O(n)$...
  ... but implementation is rather complex;
- In this course, we will see the Suffix Array;
- The Suffix Array is built in $O(n \log n)$...
  ... but the implementation is very simple!

I encourage you to study the implementation of the suffix tree by yourself!

Suffix Trie: Suffix Array (2)

- To make a Suffix array, make an array of all possible suffixes of T, and sort it;
- The order of the suffix array is the visit in preorder of the suffix tree;
- We can adapt all algorithms accordingly;

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

Sort $\rightarrow$

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: Implementation (1)

Simple Implementation

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
char T[MAX_N]; int SA[MAX_N],i,n;

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

int main() {
  n = (int) strlen (gets(T));
  for (int i = 0; i < n; i++) SA[i] = i;
  sort (SA, SA+n, cmp); // O(n^2 log n) }
```

This implementation is too slow for strings bigger than 1000 characters.

Suffix Array: Implementation (2.1)

O(n log n) implementation using “ranking pairs/radix sort”

```
char T[MAX_N]; int n; int c[MAX_N];
int RA[MAX_N], tempRA[MAX_N], SA[MAX_N], tempSA[MAX_N];

void countingSort(int k) {
  int i, sum, maxi = max(300,n); //255 ASCII chars or n
  memset(c, 0, sizeof(c));
  for (i = 0; i < n; i++) c[i+k<n? RA[i+k] : 0]++
  for (i = sum = 0; i < maxi; i++)
    { int t = c[i]; c[i] = sum; sum += t;} //frequency
  for (i = 0; i < n; i++)
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++) // update suffix array
    SA[i] = tempSA[i];
}

// ... continues next slide
```

Suffix Array: Implementation (2.2)

O(n log n) implementation using “ranking pairs/radix sort”

```
// ... continued from last slide

void constructSA() {
  int i, k, r;
  for (i = 0; i < n; i++) { RA[i] = T[i]; SA[i] = i;}
  for (k = 1; k < n; k <<=1) {
    countingSort(k); countingSort(0);
    tempRA[SA[0]] = r = 0;
    for (i = 1; i < n; i++) tempRA[SA[i]] =
           (RA[SA[i]] == RA[SA[i-1]] &&
            RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;
}}
```

Suffix Array: Using Suffix Array (1)

String Matching: Finding 'GA'

- Do a binary search once to find the lower bound;
- Do a binary search once to fint the upper bound;

Finding lower bound

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

Finding upper bound

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

Suffix Array: Using Suffix Array (2)

Longest Repeated Substring

Find the longest common prefix between suffix i and $i+1$

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$

Suffix Array: Using Suffix Array (3)

Longest Common Substring

- Create Suffix Array for appended strings *MN*;
- Find the longest common prefix that has both string enders;

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#

Problems for this Week

- Immediate Decodability
- Caesar Cypher
- Power Strings
- Where's Waldorf
- Extend to Palindrome
- String Partition
- Prince and Princess
- Power Strings
- Life Forms

Immediate Decodability

Outline

A set of tokens is decodable if it is **impossible** to write a string that can be parsed in more than one way.

Decodability is detected by checking if a token is a prefix of another.

```
Input:                          Output:
 01                             decodable
 10                             (no string is a prefix)
 0001
 00101

 001                            not decodable
 0100                           (001 is a prefix of 00101)
 00101
```

Caesar Cypher

Outline

A **k-rotation cypher** replaces every symbol N with symbol $N + k$, including spaces (which are symbol 0).

- **Input:** A list of correct words, and an encrypted text
- **Task:** Find the shift that matches the maximum number of words in the dictionary. Output the decrypted text

Notes about the problem:

- Input: Small, No case, no symbols, spaces
- Crypto text may contain words not in dictionary
- Output requirements (linebreak at 60 characters)

```
THIS      DAWN      THAT      || BUUBDLA PSSPABUAEBXO
THE       ZORRO     OTHER     || ATTACK ZORRO AT DAWN
```

Power Strings

Problem Outline

You are given a string s, and you must find the smallest string s', so that $s = s' + s' + s' + \ldots = (s')^n$. This is equal to finding s' with maximum n.

Example Input and Output:

```
INPUT          MINIMUM STRING   N
abcd           abcd             1: abcd
abababab       ab               4: ab + ab + ab + ab
kallakalla     kalla            2: kalla + kalla
abababa        abababa          1: abababa
```

This is a mixture of search and string matching. If your search is not very good, you may face TLE, so write your search carefully.

Where is Waldorf?

Problem Outline

This the traditional magazine challenge: Find words inside a cube of letters. Pay attention:

- Words can be vertical, horizontal or diagonal;
- Words can be backwards;
- Search is **not case sensitive**

```
abcDEFGhigg          Words:
hEbkWalDork          Waldorf -- 2 5
FtyAwaldORm          Bambi   -- 2 3
FtsimrLqsrc          Betty   -- 1 2
byoArBeDeyv          Dagbert -- ? ?
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
```

Extend to Palindrome

Problem Outline

You receive a word as input, and you must add the smallest number of letters at the end to make it a palindrome. Examples:

```
alert:      alertrela
abcba:      abcba
aaaalll:    aaaalllaaaa
```

Hints:

- Which letters do you add to a word to make it a palindrome;
- How do you decide if you add a letter or not?
- Can you modify the KLM algorithm to help you make the decision?

String Partition

Problem Outline

You must break a large string of digits into smaller numbers (max size: 32 bit signed integer), so that the sum of the numbers is the largest.

Hints:

- The max number of digits in the string is N=500;
- Start with a search on the breaking points;
- The total sum can be bigger than signed int;

INPUT	OUTPUT
1234554321	1234554321
5432112345	543211239
000	0
111111111111111111111111111111	3333333333

Prince and Princess

Problem outline

The Prince and the princess make different paths through the same $n * n$ grid. Both paths start and end at the same square.

Your task is to make both paths identical by eliminating steps, and print the size of the common path.

```
Input                     Output
1 7 5 4 8 3 9             4
1 4 3 5 6 2 8 9           (Common path: 1,5,8,9)

1 3 4 2 5 8 7 10          5
1 5 8 9 3 2 7 10          (Common path: 1,5,8,7,10)
```

Life Forms

Problem Outline

Given a set of strings, find the **largest common substring** that is shared by more than half of the strings.

Hints:

- Generalization of LCS, but for multiple strings;
- If there are multiple substrings of the same size, output all of them;

INPUT	OUTPUT
abcdefg	bcdefg
bcdefgh	cdefgh
cdefghi	

About these Slides

These slides were made by Claus Aranha, 2021. You are welcome to copy, re-use and modify this material.

Individual images in some slides might have been made by other authors. Please see the following pages for details.

Image Credits I

[Page 33] Suffix Tree/Array images from Steven Halim, "Competitive Programming 3", chapter 6.6