Chapter 6

String Processing

The Human Genome has approximately 3.2 Giga base pairs
— **Human Genome Project**

6.1 Overview and Motivation

In this chapter, we present one more topic that is tested in ICPC—although not as frequent[1] as graph and mathematics problems—namely: String processing. String processing is common in the research field of *bioinformatics*. As the strings (e.g. DNA strings) that researchers deal with are usually (very) long, efficient string-specific data structures and algorithms are necessary. Some of these problems are presented as contest problems in ICPCs. By mastering the content of this chapter, ICPC contestants will have a better chance at tackling those string processing problems.

String processing tasks also appear in IOI, but usually they do not require advanced string data structures or algorithms due to syllabus [20] restriction. Additionally, the input and output format of IOI tasks are usually simple[2]. This eliminates the need to code tedious input parsing or output formatting commonly found in ICPC problems. IOI tasks that require string processing are usually still solvable using the problem solving paradigms mentioned in Chapter 3. It is sufficient for IOI contestants to skim through all sections in this chapter except Section 6.5 about string processing with DP. However, we believe that it may be advantageous for IOI contestants to learn some of the more advanced materials outside of their syllabus ahead of time.

This chapter is structured as follows: It starts with an overview of basic string processing skills and a *long* list of Ad Hoc string problems solvable with that basic string processing skills. Although the Ad Hoc string problems constitute the majority of the problems listed in this chapter, we have to make a remark that recent contest problems in ACM ICPC (and also IOI) usually do not ask for basic string processing solutions *except* for the 'giveaway' problem that most teams (contestants) should be able to solve. The more important sections are the string matching problems (Section 6.4), string processing problems solvable with Dynamic Programming (DP) (Section 6.5), and finally an extensive discussion on string processing problems where we have to deal with reasonably **long** strings (Section 6.6). The last section involves a discussion on an efficient data structure for strings like Suffix **Trie**, Suffix **Tree**, and Suffix **Array**.

[1]One potential reason: String input is harder to parse correctly and string output is harder to format correctly, making such string-based I/O less preferred over the more precise integer-based I/O.

[2]IOI 2010-2012 require contestants to implement functions instead of coding I/O routines.

6.2 Basic String Processing Skills

We begin this chapter by listing several *basic* string processing skills that every competitive programmer must have. In this section, we give a series of mini tasks that you should solve one after another without skipping. You can use any of the three programming languages: C, C++, and/or Java. Try your best to come up with the shortest, most efficient implementation that you can think of. Then, compare your implementations with ours (see the answers at the back of this chapter). If you are not surprised with any of our implementations (or can even give simpler implementations), then you are already in a good shape for tackling various string processing problems. Go ahead and read the next sections. Otherwise, please spend some time studying our implementations.

1. Given a text file that contains only alphabet characters `[A-Za-z]`, digits `[0-9]`, space, and period (`'.'`), write a program to read this text file line by line until we encounter a line that *starts with* seven periods (`''.......''`). Concatenate (combine) each line into one long string `T`. When two lines are combined, give one space between them so that the last word of the previous line is separated from the first word of the current line. There can be up to 30 characters per line and no more than 10 lines for this input block. There is no trailing space at the end of each line and each line ends with a newline character. Note: The sample input text file `'ch6.txt'` is shown inside a box after question 1.(d) and before task 2.

   (a) Do you know how to store a string in your favorite programming language?

   (b) How to read a given text input line by line?

   (c) How to concatenate (combine) two strings into a larger one?

   (d) How to check if a line starts with a string `'.......'` to stop reading input?

```
I love CS3233 Competitive
Programming. i also love
AlGoRiThM
.......you must stop after reading this line as it starts with 7 dots
after the first input block, there will be one looooooooooong line...
```

2. Suppose that we have one long string `T`. We want to check if another string `P` can be found in `T`. Report all the indices where `P` appears in `T` or report `-1` if `P` cannot be found in `T`. For example, if `T = ''I love CS3233 Competitive Programming.  i also love AlGoRiThM''` and `P = 'I'`, then the output is only {0} (0-based indexing). If uppercase `'I'` and lowercase `'i'` are considered different, then the character `'i'` at index {39} is not part of the output. If `P = 'love'`, then the output is {2, 46}. If `P = 'book'`, then the output is {-1}.

   (a) How to find the first occurrence of a substring in a string (if any)?
   Do we need to implement a string matching algorithm (e.g. Knuth-Morris-Pratt algorithm discussed in Section 6.4, etc) or can we just use library functions?

   (b) How to find the next occurrence(s) of a substring in a string (if any)?

3. Suppose we want to do some simple analysis of the characters in `T` and also to transform each character in `T` into lowercase. The required analysis are: How many digits, vowels `[aeiouAEIOU]`, and consonants (other alphabets that are not vowels) are there in `T`? Can you do all these in $O(n)$ where n is the length of the string `T`?

4. Next, we want to break this one long string `T` into *tokens* (substrings) and store them into an array of strings called `tokens`. For this mini task, the *delimiters* of these tokens are spaces and periods (thus breaking sentences into words). For example, if we *tokenize* the string `T` (in lowercase), we will have these `tokens = {'i', 'love', 'cs3233', 'competitive', 'programming', 'i', 'also', 'love', 'algorithm'}`. Then, we want to sort this array of strings lexicographically[3] and then find the lexicographically smallest string. That is, we have sorted `tokens`: `{'algorithm', 'also', 'competitive', 'cs3233', 'i', 'i', 'love', 'love', 'programming'}`. Thus, the lexicographically smallest string for this example is `'algorithm'`.
   (a) How to tokenize a string?
   (b) How to store the tokens (the shorter strings) in an *array* of strings?
   (c) How to sort an array of strings lexicographically?
5. Now, identify which word appears the most in `T`. In order to answer this query, we need to count the frequency of each word. For `T`, the output is either `'i'` or `'love'`, as both appear twice. Which data structure should be used for this mini task?
6. The given text file has one more line after a line that starts with `'.......'` but the length of this last line is not constrained. Your task is to count how many characters there are in the last line. How to read a string if its length is not known in advance?

Tasks and Source code: `ch6_01_basic_string.html/cpp/java`

Profile of Algorithm Inventors

Donald Ervin Knuth (born 1938) is a computer scientist and Professor Emeritus at Stanford University. He is the author of the popular Computer Science book: *"The Art of Computer Programming"*. Knuth has been called the 'father' of the analysis of algorithms. Knuth is also the creator of the TEX, the computer typesetting system used in this book.

James Hiram Morris (born 1941) is a Professor of Computer Science. He is a co-discoverer of the Knuth-Morris-Pratt algorithm for string-search.

Vaughan Ronald Pratt (born 1944) is a Professor Emeritus at Stanford University. He was one of the earliest pioneers in the field of computer science. He has made several contributions to foundational areas such as search algorithms, sorting algorithms, and primality testing. He is also a co-discoverer of the Knuth-Morris-Pratt algorithm for string-search.

Saul B. Needleman and **Christian D. Wunsch** jointly published the string alignment Dynamic Programming algorithm in 1970. Their DP algorithm is discussed in this book.

Temple F. Smith is a Professor in biomedical engineering who helped to develop the Smith-Waterman algorithm developed with Michael Waterman in 1981. The Smith-Waterman algorithm serves as the basis for multi sequence comparisons, identifying the segment with the maximum *local* sequence similarity for identifying similar DNA, RNA, and protein segments.

Michael S. Waterman is a Professor at the University of Southern California. Waterman is one of the founders and current leaders in the area of computational biology. His work has contributed to some of the most widely-used tools in the field. In particular, the Smith-Waterman algorithm (developed with Temple F. Smith) is the basis for many sequence comparison programs.

[3]Basically, this is a sort order like the one used in our common dictionary.

6.3 Ad Hoc String Processing Problems

Next, we continue our discussion with something light: The Ad Hoc string processing problems. They are programming contest problems involving strings that require no more than basic programming skills and perhaps some basic string processing skills discussed in Section 6.2 earlier. We only need to read the requirements in the problem description carefully and code the usually short solution. Below, we give a list of such Ad Hoc string processing problems with hints. These programming exercises have been further divided into sub-categories.

- Cipher/Encode/Encrypt/Decode/Decrypt
  It is everyone's wish that their private digital communications are secure. That is, their (string) messages can only be read by the intended recipient(s). Many ciphers have been invented for this purpose and many (of the simpler ones) end up as Ad Hoc programming contest problems, each with its own encoding/decoding rules. There are many such problems in UVa online judge [47]. Thus, we have further split this category into two: the easier versus the harder ones. Try solving some of them, especially those that we classify as **`must try *`**. It is interesting to learn a bit about *Computer Security/Cryptography* by solving these problems.

- Frequency Counting
  In this group of problems, the contestants are asked to count the frequency of a letter (easy, use Direct Addressing Table) or a word (harder, the solution is either using a balanced Binary Search Tree—like C++ STL `map`/Java `TreeMap`—or Hash table). Some of these problems are actually related to Cryptography (the previous sub-category).

- Input Parsing
  This group of problems is not for IOI contestants as the IOI syllabus enforces the input of IOI tasks to be formatted as simple as possible. However, there is no such restriction in ICPC. Parsing problems range from the simpler ones that can be dealt with an iterative parser and the more complex ones involving some grammars that requires recursive descent parser or Java String/Pattern class.

- Solvable with Java String/Pattern class (Regular Expression)

  Some (but rare) string processing problems are solvable with one liner[4] code that use `matches(String regex)`, `replaceAll(String regex, String replacement)`, and/or other useful functions of Java `String` class. To be able to do this, one has to master the concept of Regular Expression (Regex). We will not discuss Regex in detail but we will show two usage examples:

  1. In UVa 325 - Identifying Legal Pascal Real Constants, we are asked to decide if the given line of input is a legal Pascal Real constant. Suppose the line is stored in `String s`, then the following one-liner Java code is the required solution:

     ```
     s.matches("[-+]?\\d+(\\.\\d+([eE][-+]?\\d+)?|[eE][-+]?\\d+)")
     ```

  2. In UVa 494 - Kindergarten Counting Game, we are asked to count how many words are there in a given line. Here, a word is defined as a consecutive sequence of letters (upper and/or lower case). Suppose the line is stored in `String s`, then the following one-liner Java code is the required solution:

     ```
     s.replaceAll("[^a-zA-Z]+", " ").trim().split(" ").length
     ```

[4]We can solve these problems without Regular Expression, but the code may be longer.

- Output Formatting
  This is another group of problems that is also not for IOI contestants. This time, the output is the problematic one. In an ICPC problem set, such problems are used as 'coding warm up' or the 'time-waster problem' for the contestants. Practice your coding skills by solving these problems *as fast as possible* as such problems can differentiate the penalty time for each team.
- String Comparison
  In this group of problems, the contestants are asked to compare strings with various criteria. This sub-category is similar to the string matching problems in the next section, but these problems mostly use `strcmp`-related functions.
- Just Ad Hoc
  These are other Ad Hoc string related problems that cannot be classified as one of the other sub categories above.

Programming Exercises related to Ad Hoc String Processing:

- Cipher/Encode/Encrypt/Decode/Decrypt, Easier
  1. UVa 00245 - Uncompress (use the given algorithm)
  2. UVa 00306 - Cipher (can be made faster by avoiding cycle)
  3. UVa 00444 - Encoder and Decoder (each char is mapped to 2 or 3 digits)
  4. UVa 00458 - The Decoder (shift each character's ASCII value by -7)
  5. UVa 00483 - Word Scramble (read char by char from left to right)
  6. UVa 00492 - Pig Latin (ad hoc, similar to UVa 483)
  7. UVa 00641 - Do the Untwist (reverse the given formula and simulate)
  8. UVa 00739 - Soundex Indexing (straightforward conversion problem)
  9. UVa 00795 - Sandorf's Cipher (prepare an 'inverse mapper')
  10. UVa 00865 - Substitution Cypher (simple character substitution mapping)
  11. UVa 10019 - Funny Encryption Method (not hard, find the pattern)
  12. UVa 10222 - Decode the Mad Man (simple decoding mechanism)
  13. **UVa 10851 - 2D Hieroglyphs ... *** (ignore border; treat '\/' as 1/0; read from bottom)
  14. **UVa 10878 - Decode the Tape *** (treat space/'o' as 0/1, then it is binary to decimal conversion)
  15. UVa 10896 - Known Plaintext Attack (try all possible keys; use tokenizer)
  16. UVa 10921 - Find the Telephone (simple conversion problem)
  17. UVa 11220 - Decoding the message (follow instruction in the problem)
  18. **UVa 11278 - One-Handed Typist *** (map QWERTY keys to DVORAK)
  19. UVa 11541 - Decoding (read char by char and simulate)
  20. UVa 11716 - Digital Fortress (simple cipher)
  21. UVa 11787 - Numeral Hieroglyphs (follow the description)
  22. UVa 11946 - Code Number (ad hoc)
- Cipher/Encode/Encrypt/Decode/Decrypt, Harder
  1. UVa 00213 - Message Decoding (decrypt the message)
  2. UVa 00468 - Key to Success (letter frequency mapping)
  3. ***UVa 00554 - Caesar Cypher **** (try all shifts; output formatting)

4. *UVa 00632 - Compression (II)* (simulate the process, use sorting)
5. *UVa 00726 - Decode* (frequency cypher)
6. UVa 00740 - Baudot Data ... (just simulate the process)
7. UVa 00741 - Burrows Wheeler Decoder (simulate the process)
8. UVa 00850 - Crypt Kicker II (plaintext attack, tricky test cases)
9. UVa 00856 - The Vigenère Cipher (3 nested loops; one for each digit)
10. **UVa 11385 - Da Vinci Code *** (string manipulation + Fibonacci)
11. **UVa 11697 - Playfair Cipher *** (follow the description, a bit tedious)

- Frequency Counting
  1. UVa 00499 - What's The Frequency ... (use 1D array for frequency counting)
  2. UVa 00895 - Word Problem (get the letter frequency of each word, compare with puzzle line)
  3. **UVa 00902 - Password Search *** (read char by char; count word freq)
  4. UVa 10008 - What's Cryptanalysis? (character frequency count)
  5. UVa 10062 - Tell me the frequencies (ASCII character frequency count)
  6. **UVa 10252 - Common Permutation *** (count freq of each alphabet)
  7. UVa 10293 - Word Length and Frequency (straightforward)
  8. UVa 10374 - Election (use `map` for frequency counting)
  9. UVa 10420 - List of Conquests (word frequency counting, use `map`)
  10. UVa 10625 - GNU = GNU'sNotUnix (frequency addition n times)
  11. UVa 10789 - Prime Frequency (check if a letter's frequency is prime)
  12. **UVa 11203 - Can you decide it ... *** (problem description is convoluted, but this problem is actually easy)
  13. UVa 11577 - Letter Frequency (straightforward problem)

- Input Parsing (Non Recursive)
  1. UVa 00271 - Simply Syntax (grammar check, linear scan)
  2. UVa 00327 - Evaluating Simple C ... (implementation can be tricky)
  3. UVa 00391 - Mark-up (use flags, tedious parsing)
  4. UVa 00397 - Equation Elation (iteratively perform the next operation)
  5. UVa 00442 - Matrix Chain Multiplication (properties of matrix chain mult)
  6. UVa 00486 - English-Number Translator (parsing)
  7. UVa 00537 - Artificial Intelligence? (simple formula; parsing is difficult)
  8. UVa 01200 - A DP Problem (LA 2972, Tehran03, tokenize linear equation)
  9. ***UVa 10906 - Strange Integration **** (BNF parsing, iterative solution)
  10. UVa 11148 - Moliu Fractions (extract integers, simple/mixed fractions from a line; a bit of gcd—see Section 5.5.2)
  11. ***UVa 11357 - Ensuring Truth **** (the problem description looks scary—a SAT (satisfiability) problem; the presence of BNF grammar makes one think of recursive descent parser; however, only one clause needs to be satisfied to get TRUE; a clause can be satisfied if for all variables in the clause, its inverse is not in the clause too; now, we have a much simpler problem)
  12. **UVa 11878 - Homework Checker *** (mathematical expression parsing)
  13. *UVa 12543 - Longest Word* (LA6150, HatYai12, iterative parser)

- Input Parsing (Recursive)
  1. UVa 00384 - Slurpys (recursive grammar check)
  2. UVa 00464 - Sentence/Phrase Generator (generate output based on the given BNF grammar)
  3. UVa 00620 - Cellular Structure (recursive grammar check)
  4. **UVa 00622 - Grammar Evaluation *** (recursive BNF grammar check/evaluation)
  5. UVa 00743 - The MTM Machine (recursive grammar check)
  6. **UVa 10854 - Number of Paths *** (recursive parsing plus counting)
  7. *UVa 11070 - The Good Old Times* (recursive grammar evaluation)
  8. ***UVa 11291 - Smeech **** (recursive descent parser)
- Solvable with Java String/Pattern class (Regular Expression)
  1. **UVa 00325 - Identifying Legal ... *** (see the Java solution above)
  2. **UVa 00494 - Kindergarten Counting ... *** (see the Java solution above)
  3. UVa 00576 - Haiku Review (parsing, grammar)
  4. **UVa 10058 - Jimmi's Riddles *** (solvable with Java regular expression)
- Output Formatting
  1. UVa 00110 - Meta-loopless sort (actually an ad hoc sorting problem)
  2. *UVa 00159 - Word Crosses* (tedious output formatting problem)
  3. UVa 00320 - Border (requires flood fill technique)
  4. *UVa 00330 - Inventory Maintenance* (use `map` to help)
  5. *UVa 00338 - Long Multiplication* (tedious)
  6. *UVa 00373 - Romulan Spelling* (check 'g' versus 'p', ad hoc)
  7. *UVa 00426 - Fifth Bank of ...* (tokenize; sort; reformat output)
  8. UVa 00445 - Marvelous Mazes (simulation, output formatting)
  9. **UVa 00488 - Triangle Wave *** (use several loops)
  10. UVa 00490 - Rotating Sentences (2d array manipulation, output formatting)
  11. *UVa 00570 - Stats* (use `map` to help)
  12. *UVa 00645 - File Mapping* (use recursion to simulate directory structure, it helps the output formatting)
  13. *UVa 00890 - Maze (II)* (simulation, follow the steps, tedious)
  14. UVa 01219 - Team Arrangement (LA 3791, Tehran06)
  15. *UVa 10333 - The Tower of ASCII* (a real time waster problem)
  16. UVa 10500 - Robot maps (simulate, output formatting)
  17. UVa 10761 - Broken Keyboard (tricky with output formatting; note that 'END' is part of input!)
  18. **UVa 10800 - Not That Kind of Graph *** (tedious problem)
  19. *UVa 10875 - Big Math* (simple but tedious problem)
  20. UVa 10894 - Save Hridoy (how fast can you can solve this problem?)
  21. UVa 11074 - Draw Grid (output formatting)
  22. *UVa 11482 - Building a Triangular ...* (tedious...)
  23. UVa 11965 - Extra Spaces (replace consecutive spaces with only one space)
  24. ***UVa 12155 - ASCII Diamondi **** (use proper index manipulation)
  25. *UVa 12364 - In Braille* (2D array check, check all possible digits [0..9])

- String Comparison
  1. UVa 00409 - Excuses, Excuses (tokenize and compare with list of excuses)
  2. **UVa 00644 - Immediate Decodability *** (use brute force)
  3. UVa 00671 - Spell Checker (string comparison)
  4. *UVa 00912 - Live From Mars* (simulation, find and replace)
  5. **UVa 11048 - Automatic Correction ... *** (flexible string comparison with respect to a dictionary)
  6. **UVa 11056 - Formula 1 *** (sorting, case-insensitive string comparison)
  7. UVa 11233 - Deli Deli (string comparison)
  8. UVa 11713 - Abstract Names (modified string comparison)
  9. UVa 11734 - Big Number of Teams ... (modified string comparison)
- Just Ad Hoc
  1. UVa 00153 - Permalex (find formula for this, similar to UVa 941)
  2. UVa 00263 - Number Chains (sort digits, convert to integers, check cycle)
  3. UVa 00892 - Finding words (basic string processing problem)
  4. **UVa 00941 - Permutations *** (formula to get the n-th permutation)
  5. UVa 01215 - String Cutting (LA 3669, Hanoi06)
  6. UVa 01239 - Greatest K-Palindrome ... (LA 4144, Jakarta08, brute-force)
  7. UVa 10115 - Automatic Editing (simply do what they want, uses string)
  8. *UVa 10126 - Zipf's Law* (sort the words to simplify this problem)
  9. UVa 10197 - Learning Portuguese (must follow the description very closely)
  10. UVa 10361 - Automatic Poetry (read, tokenize, process as requested)
  11. UVa 10391 - Compound Words (more like data structure problem)
  12. **UVa 10393 - The One-Handed Typist *** (follow problem description)
  13. UVa 10508 - Word Morphing (number of words = number of letters + 1)
  14. UVa 10679 - I Love Strings (the test data weak; just checking if T is a prefix of S is AC when it should not)
  15. **UVa 11452 - Dancing the Cheeky ... *** (string period, small input, BF)
  16. UVa 11483 - Code Creator (straightforward, use 'escape character')
  17. UVa 11839 - Optical Reader (illegal if mark 0 or > 1 alternatives)
  18. UVa 11962 - DNA II (find formula; similar to UVa 941; base 4)
  19. *UVa 12243 - Flowers Flourish ...* (simple string tokenizer problem)
  20. *UVa 12414 - Calculating Yuan Fen* (brute force problem involving string)

6.4 String Matching

String *Matching* (a.k.a String *Searching*[5]) is a problem of finding the starting index (or indices) of a (sub)string (called *pattern* `P`) in a longer string (called *text* `T`). Example: Let's assume that we have `T = 'STEVEN EVENT'`. If `P = 'EVE'`, then the answers are index 2 and 7 (0-based indexing). If `P = 'EVENT'`, then the answer is index 7 only. If `P = 'EVENING'`, then there is no answer (no matching found and usually we return either -1 or NULL).

6.4.1 Library Solutions

For most *pure* String Matching problems on reasonably short strings, we can just use string library in our programming language. It is **`strstr`** in C <`string.h`>, **`find`** in C++ <`string`>, **`indexOf`** in Java **`String`** class. Please revisit Section 6.2, mini task 2 that discusses these string library solutions.

6.4.2 Knuth-Morris-Pratt's (KMP) Algorithm

In Section 1.2.3, Question 7, we have an exercise of finding all the occurrences of a substring P (of length m) in a (long) string T (of length n), if any. The code snippet, reproduced below with comments, is actually the *naïve* implementation of String Matching algorithm.

```
void naiveMatching() {
  for (int i = 0; i < n; i++) {       // try all potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++)        // use boolean flag 'found'
      if (i + j >= n || P[j] != T[i + j])              // if mismatch found
        found = false;      // abort this, shift the starting index i by +1
    if (found)                                 // if P[0..m-1] == T[i..i+m-1]
      printf("P is found at index %d in T\n", i);
} }
```

This naïve algorithm can run in $O(n)$ *on average* if applied to natural text like the paragraphs of this book, but it can run in $O(nm)$ with the worst case programming contest input like this: `T = 'AAAAAAAAAAB'` ('A' ten times and then one 'B') and `P = 'AAAAB'`. The naive algorithm will keep failing at the last character of pattern `P` and then try the next starting index which is just +1 than the previous attempt. This is not efficient. Unfortunately, a good problem author will include such test case in their secret test data.

In 1977, Knuth, Morris, and Pratt—thus the name of KMP—invented a better String Matching algorithm that makes use of the information gained by previous character comparisons, especially those that matches. KMP algorithm *never* re-compares a character in `T` that has matched a character in `P`. However, it works similar to the naïve algorithm if the *first* character of pattern `P` and the current character in `T` is a mismatch. In the example below[6], comparing `P[j]` and `T[i]` and from `i` = 0 to 13 with `j = 0` (the first character of `P`) is no different than the naïve algorithm.

[5]We deal with this String Matching problem almost every time we read/edit text using computer. How many times have you pressed the well-known 'CTRL + F' button (standard Windows shortcut for the 'find feature') in typical word processing softwares, web browsers, etc?

[6]The sentence in string `T` below is just for illustration. It is not grammatically correct.

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
    0123456789012
              1
    ^ the first character of P mismatch with T[i] from index i = 0 to 13
    KMP has to shift the starting index i by +1, as with naive matching.
... at i = 14 and j = 0 ...
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =               SEVENTY SEVEN
                  0123456789012
                            1
                             ^ then mismatch at index i = 25 and j = 11
```

There are 11 matches from index i = 14 to 24, but one mismatch at i = 25 (j = 11). The naïve matching algorithm will inefficiently restart from index i = 15 but KMP can resume from i = 25. This is because the matched characters before the mismatch is ‘SEVENTY SEV’. ‘SEV’ (of length 3) appears as BOTH proper suffix and prefix of ‘SEVENTY SEV’. This ‘SEV’ is also called as the **border** of ‘SEVENTY SEV’. We can safely skip index i = 14 to 21: ‘SEVENTY ’ in ‘SEVENTY SEV’ as it will not match again, but we cannot rule out the possibility that the next match starts from the second ‘SEV’. So, KMP resets j back to 3, skipping 11 - 3 = 8 characters of ‘SEVENTY ’ (notice the trailing space), while i remains at index 25. This is the major difference between KMP and the naïve matching algorithm.

```
... at i = 25 and j = 3 (This makes KMP efficient) ...
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                       SEVENTY SEVEN
                          0123456789012
                                    1
                             ^ then immediate mismatch at index i = 25, j = 3
```

This time the prefix of P before mismatch is ‘SEV’, but it does not have a border, so KMP resets j back to 0 (or in another word, restart matching pattern P from the front again).

```
... mismatches from i = 25 to i = 29... then matches from i = 30 to i = 42 ...
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                               SEVENTY SEVEN
                                  0123456789012
                                            1
```

This is a match, so P = ‘SEVENTY SEVEN’ is found at index i = 30. After this, KMP knows that ‘SEVENTY SEVEN’ has ‘SEVEN’ (of length 5) as border, so KMP resets j back to 5, effectively skipping 13 - 5 = 8 characters of ‘SEVENTY ’ (notice the trailing space), immediately resumes the search from i = 43, and gets another match. This is efficient.

```
... at i = 43 and j = 5, we have matches from i = 43 to i = 50 ...
So P = 'SEVENTY SEVEN' is found again at index i = 38.
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                                       SEVENTY SEVEN
                                          0123456789012
                                                    1
```

To achieve such speed up, KMP has to preprocess the pattern string and get the 'reset table' `b` (back). If given pattern string P = 'SEVENTY SEVEN', then table `b` will looks like this:

```
                         1
     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
b = -1 0 0 0 0 0 0 0 0 1 2 3 4 5
```

This means, if mismatch happens in `j = 11` (see the example above), i.e. after finding matches for 'SEVENTY SEV', then we know that we have to re-try matching `P` from index `j = b[11] = 3`, i.e. KMP now assumes that it has matched only the first three characters of 'SEVENTY SEV', which is 'SEV', because the next match can start with that prefix 'SEV'. The relatively short implementation of the KMP algorithm with comments is shown below. This implementation has a time complexity of $O(n + m)$.

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N];                              // T = text, P = pattern
int b[MAX_N], n, m;     // b = back table, n = length of T, m = length of P

void kmpPreprocess() {              // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1;                           // starting values
  while (i < m) {                          // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j];  // different, reset j using b
    i++; j++;                              // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12, 13 with j = 0, 1, 2, 3, 4, 5
} }                           // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() {   // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0;                                       // starting values
  while (i < n) {                                   // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j];  // different, reset j using b
    i++; j++;                              // if same, advance both pointers
    if (j == m) {                                 // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j];                       // prepare j for the next possible match
} } }
```

Source code: `ch6_02_kmp.cpp/java`

Exercise 6.4.1*: Run `kmpPreprocess()` on `P = 'ABABA'` and show the reset table `b`!

Exercise 6.4.2*: Run `kmpSearch()` with `P = 'ABABA'` and `T = 'ACABAABABDABABA'`. Explain how the KMP search looks like?

6.4.3 String Matching in a 2D Grid

The string matching problem can also be posed in 2D. Given a 2D grid/array of characters (instead of the well-known 1D array of characters), find the occurrence(s) of pattern P in the grid. Depending on the problem requirement, the search direction can be to 4 or 8 cardinal directions, and either the pattern must be found in a straight line or it can bend. See the following example below.

```
abcdefghigg      // From UVa 10010 - Where's Waldorf?
hebkWaldork      // We can go to 8 directions, but must be straight
ftyawAldorm      // 'WALDORF' is highlighted as capital letters in the grid
ftsimrLqsrc
byoarbeDeyv      // Can you find 'BAMBI' and 'BETTY'?
klcbqwikOmk
strebgadhRb      // Can you find 'DAGBERT' in this row?
yuiqlxcnbjF
```

The solution for such string matching in a 2D grid is usually a *recursive backtracking* (see Section 3.2.2). This is because unlike the 1D counterpart where we always go to the right, at every coordinate (row, col) of the 2D grid, we have *more than one choice* to explore.

To speed up the backtracking process, usually we employ this simple pruning strategy: Once the recursion depth exceeds the length of pattern P, we can immediately prune that recursive branch. This is also called as *depth-limited search* (see Section 8.2.5).

Programming Exercises related to String Matching

- Standard
  1. UVa 00455 - Periodic String (find s in s + s)
  2. *UVa 00886 - Named Extension Dialing* (convert first letter of given name and all the letters of the surname into digits; then do a kind of special string matching where we want the matching to start at the *prefix* of a string)
  3. **UVa 10298 - Power Strings *** (find s in s + s, similar to UVa 455)
  4. UVa 11362 - Phone List (string sort, matching)
  5. **UVa 11475 - Extend to Palindromes *** ('border' of KMP)
  6. **UVa 11576 - Scrolling Sign *** (modified string matching; complete search)
  7. UVa 11888 - Abnormal 89's (to check 'alindrome', find reverse of s in s + s)
  8. *UVa 12467 - Secret word* (similar idea with UVa 11475, if you can solve that problem, you should be able to solve this problem)
- In 2D Grid
  1. **UVa 00422 - Word Search Wonder *** (2D grid, backtracking)
  2. *UVa 00604 - The Boggle Game* (2D matrix, backtracking, sort, and compare)
  3. *UVa 00736 - Lost in Space* (2D grid, a bit modified)
  4. **UVa 10010 - Where's Waldorf? *** (discussed in this section)
  5. **UVa 11283 - Playing Boggle *** (2D grid, backtracking, do not count twice)

6.5 String Processing with Dynamic Programming

In this section, we discuss several string processing problems that are solvable with DP technique discussed in Section 3.5. The first two (String Alignment and Longest Common Subsequence) are *classical* problems and should be known by all competitive programmers. Additionally, we have added a collection of some known twists of these problems.

An important note: For various DP problems on string, we usually manipulate the *integer indices* of the strings and not the actual strings (or substrings) themselves. Passing substrings as parameters of recursive functions is strongly not recommended as it is very slow and hard to memoize.

6.5.1 String Alignment (Edit Distance)

The String Alignment (or Edit Distance[7]) problem is defined as follows: Align[8] two strings `A` with `B` with the maximum alignment score (or minimum number of edit operations):

After aligning `A` with `B`, there are a few possibilities between character `A[i]` and `B[i]`:
1. Character `A[i]` and `B[i]` **match** and we do nothing (assume this worth '+2' score),
2. Character `A[i]` and `B[i]` **mismatch** and we replace `A[i]` with `B[i]` (assume '-1' score),
3. We insert a space in `A[i]` (also '-1' score),
4. We delete a letter from `A[i]` (also '-1' score).

For example: (note that we use a special symbol '_' to denote a space)

```
A = 'ACAATCC' -> 'A_CAATCC'           // Example of a non optimal alignment
B = 'AGCATGC' -> 'AGCATGC_'                  // Check the optimal one below
                  2-22--2-              // Alignment Score = 4*2 + 4*-1 = 4
```

A brute force solution that tries all possible alignments will get TLE even for medium-length strings `A` and/or `B`. The solution for this problem is the Needleman-Wunsch's (bottom-up) DP algorithm [62]. Consider two strings `A[1..n]` and `B[1..m]`. We define $V(i, j)$ to be the score of the optimal alignment between prefix `A[1..i]` and `B[1..j]` and $score(C1, C2)$ is a function that returns the score if character $C1$ is aligned with character $C2$.

Base cases:
$V(0, 0) = 0$ // no score for matching two empty strings
$V(i, 0) = i \times score(A[i], \_)$ // delete substring `A[1..i]` to make the alignment, $i > 0$
$V(0, j) = j \times score(\_, B[j])$ // insert spaces in `B[1..j]` to make the alignment, $j > 0$

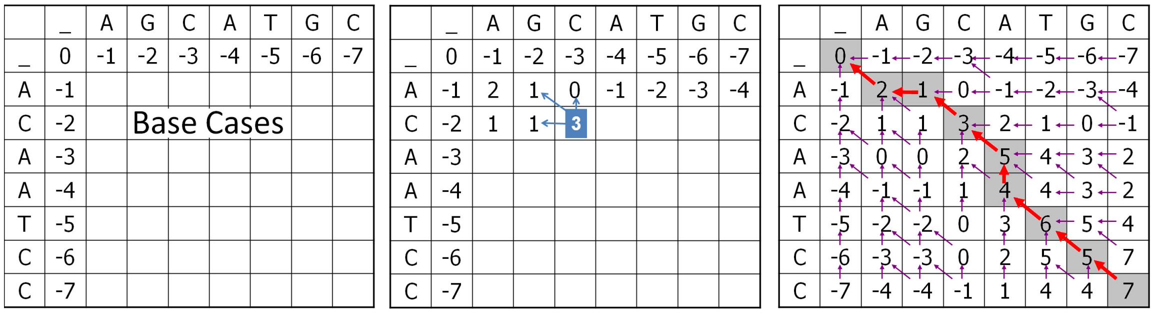
Recurrences: For $i > 0$ and $j > 0$:
$V(i, j) = max(option1, option2, option3)$, where
$option1 = V(i - 1, j - 1) + score(A[i], B[j])$ // score of match or mismatch
$option2 = V(i - 1, j) + score(A[i], \_)$ // delete A_i
$option3 = V(i, j - 1) + score(\_, B[j])$ // insert B_j

In short, this DP algorithm concentrates on the three possibilities for the last pair of characters, which must be either a match/mismatch, a deletion, or an insertion. Although we do not know which one is the best, we can try all possibilities while avoiding the re-computation of overlapping subproblems (i.e. basically a DP technique).

[7]Another name for 'edit distance' is 'Levenshtein Distance'. One notable application of this algorithm is the spelling checker feature commonly found in popular text editors. If a user misspells a word, like 'probelm', then a clever text editor that realizes that this word has a very close edit distance to the correct word 'problem' can do the correction automatically.

[8]Align is a process of inserting spaces to strings A or B such that they have the same number of characters. You can view 'inserting spaces to B' as 'deleting the corresponding aligned characters of A'.

```
A = 'xxx...xx'      A = 'xxx...xx'      A = 'xxx...x_'
            |                   |                   |
B = 'yyy...yy'      B = 'yyy...y_'      B = 'yyy...yy'
match/mismatch      delete              insert
```



	_	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							

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

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

Figure 6.1: Example: `A = 'ACAATCC'` and `B = 'AGCATGC'` (alignment score = 7)

With a simple scoring function where a match gets a +2 points and mismatch, insert, delete all get a -1 point, the detail of string alignment score of `A = 'ACAATCC'` and `B = 'AGCATGC'` is shown in Figure 6.1. Initially, only the base cases are known. Then, we can fill the values row by row, left to right. To fill in $V(i, j)$ for $i, j > 0$, we just need three other values: $V(i-1, j-1)$, $V(i-1, j)$, and $V(i, j-1)$—see Figure 6.1, middle, row 2, column 3. The maximum alignment score is stored at the bottom right cell (7 in this example).

To reconstruct the solution, we follow the darker cells from the bottom right cell. The solution for the given strings `A` and `B` is shown below. Diagonal arrow means a match or a mismatch (e.g. the last character `..C`). Vertical arrow means a deletion (e.g. `..CAA..` to `..C_A..`). Horizontal arrow means an insertion (e.g. `A_C..` to `AGC..`).

```
A = 'A_CAAT[C]C'                                    // Optimal alignment
B = 'AGC_AT[G]C'                      // Alignment score = 5*2 + 3*-1 = 7
```

The space complexity of this (bottom-up) DP algorithm is $O(nm)$—the size of the DP table. We need to fill in all cells in the table in $O(1)$ per cell. Thus, the time complexity is $O(nm)$.

Source code: `ch6_03_str_align.cpp/java`

Exercise 6.5.1.1: Why is the cost of a match +2 and the costs of replace, insert, delete are all -1? Are they magic numbers? Will +1 for match work? Can the costs for replace, insert, delete be different? Restudy the algorithm and discover the answer.

Exercise 6.5.1.2: The example source code: `ch6_03_str_align.cpp/java` only show the optimal alignment *score*. Modify the given code to actually show the *actual alignment*!

Exercise 6.5.1.3: Show how to use the 'space saving trick' shown in Section 3.5 to improve this Needleman-Wunsch's (bottom-up) DP algorithm! What will be the new space and time complexity of your solution? What is the drawback of using such a formulation?

Exercise 6.5.1.4: The String Alignment problem in this section is called the **global** alignment problem and runs in $O(nm)$. If the given contest problem is limited to d insertions or deletions only, we can have a faster algorithm. Find a simple tweak to the Needleman-Wunsch's algorithm so that it performs at most d insertions or deletions and runs faster!

Exercise 6.5.1.5: Investigate the improvement of Needleman-Wunsch's algorithm (the **Smith-Waterman's** algorithm [62]) to solve the **local** alignment problem!

6.5.2 Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is defined as follows: Given two strings A and B, what is the longest common subsequence between them. For example, `A = 'ACAATCC'` and `B = 'AGCATGC'` have LCS of length 5, i.e. `'ACATC'`.

This LCS problem can be reduced to the String Alignment problem presented earlier, so we can use the same DP algorithm. We set the cost for mismatch as negative infinity (e.g. -1 Billion), cost for insertion and deletion as 0, and the cost for match as 1. This makes the Needleman-Wunsch's algorithm for String Alignment to never consider mismatches.

Exercise 6.5.2.1: What is the LCS of `A = 'apple'` and `B = 'people'`?

Exercise 6.5.2.2: The Hamming distance problem, i.e. finding the number of different characters between two equal-length strings, can be reduced to String Alignment problem. Assign an appropriate cost to match, mismatch, insert, and delete so that we can compute the Hamming distance between two strings using Needleman-Wunsch's algorithm!

Exercise 6.5.2.3: The LCS problem can be solved in $O(n \log k)$ when all characters are distinct, e.g. if you are given two permutations as in UVa 10635. Solve this variant!

6.5.3 Non Classical String Processing with DP

UVa 11151 - Longest Palindrome

A palindrome is a string that can be read the same way in either direction. Some variants of palindrome finding problems are solvable with DP technique, e.g. UVa 11151 - Longest Palindrome: Given a string of up to $n = 1000$ characters, determine the length of the longest palindrome that you can make from it by deleting zero or more characters. Examples:

'ADAM' → 'ADA' (of length 3, delete 'M')
'MADAM' → 'MADAM' (of length 5, delete nothing)
'NEVERODDOREVENING' → 'NEVERODDOREVEN' (of length 14, delete 'ING')
'RACEF1CARFAST' → 'RACECAR' (of length 7, delete 'F1' and 'FAST')

The DP solution: let $len(l, r)$ be the length of the longest palindrome from string `A[l..r]`.

Base cases:
If $(l = r)$, then $len(l, r) = 1$. // odd-length palindrome
If $(l + 1 = r)$, then $len(l, r) = 2$ if $(A[l] = A[r])$, or 1 otherwise. // even-length palindrome

Recurrences:
If $(A[l] = A[r])$, then $len(l, r) = 2 + len(l + 1, r - 1)$. // both corner characters are the same
else $len(l, r) = max(len(l, r - 1), len(l + 1, r))$. // increase left side or decrease right side

This DP solution has time complexity of $O(n^2)$.

Exercise 6.5.3.1*: Can we use the Longest Common Subsequence solution shown in Section 6.5.2 to solve UVa 11151? If we can, how? What is the time complexity?

Exercise 6.5.3.2*: Suppose that we are now interested to find the longest palindrome in a given string with length up to $n = 10000$ characters. This time, we are not allowed to delete any character. What should be the solution?

Programming Exercises related to String Processing with DP:

- Classic
  1. UVa 00164 - String Computer (String Alignment/Edit Distance)
  2. **UVa 00526 - Edit Distance *** (String Alignment/Edit Distance)
  3. UVa 00531 - Compromise (Longest Common Subsequence; print the solution)
  4. UVa 01207 - AGTC (LA 3170, Manila06, classical String Edit problem)
  5. UVa 10066 - The Twin Towers (Longest Common Subsequence problem, but not on 'string')
  6. UVa 10100 - Longest Match (Longest Common Subsequence)
  7. **UVa 10192 - Vacation *** (Longest Common Subsequence)
  8. UVa 10405 - Longest Common ... (Longest Common Subsequence)
  9. **UVa 10635 - Prince and Princess *** (find LCS of two permutations)
  10. UVa 10739 - String to Palindrome (variation of edit distance)
- Non Classic
  1. *UVa 00257 - Palinwords* (standard DP palindrome plus brute force checks)
  2. *UVa 10453 - Make Palindrome* (s: (L, R); t: (L+1, R-1) if S[L] == S[R]; or one plus min of(L + 1, R) or (L, R - 1); also print the required solution)
  3. UVa 10617 - Again Palindrome (manipulate indices, not the actual string)
  4. ***UVa 11022 - String Factoring **** (s: the min weight of substring `[i..j]`)
  5. **UVa 11151 - Longest Palindrome *** (discussed in this section)
  6. **UVa 11258 - String Partition *** (discussed in this section)
  7. *UVa 11552 - Fewest Flops* (`dp(i, c)` = minimum number of chunks after considering the first `i` segments ending with character `c`)

Profile of Algorithm Inventors

Udi Manber is an Israeli computer scientist. He works in Google as one of their vice presidents of engineering. Along with Gene Myers, Manber invented Suffix Array data structure in 1991.

Eugene "Gene" Wimberly Myers, Jr. is an American computer scientist and bioinformatician, who is best known for his development of the BLAST (Basic Local Alignment Search Tool) tool for sequence analysis. His 1990 paper that describes BLAST has received over 24000 citations making it among the most highly cited paper ever. He also invented Suffix Array with Udi Manber.

6.6 Suffix Trie/Tree/Array

Suffix Trie, Suffix Tree, and Suffix Array are efficient and related data structures for strings. We do not discuss this topic in Section 2.4 as these data structures are unique to strings.

6.6.1 Suffix Trie and Applications

The **suffix** i (or the i-th suffix) of a string is a 'special case' of substring that goes from the i-th character of the string up to the *last* character of the string. For example, the 2-th suffix of 'STEVEN' is 'EVEN', the 4-th suffix of 'STEVEN' is 'EN' (0-based indexing).

A **Suffix Trie**[9] of a set of strings S is a tree of all possible suffixes of strings in S. Each edge label represents a character. Each vertex represents a suffix indicated by its path label: A sequence of edge labels from root to that vertex. Each vertex is connected to (some of) the other 26 vertices (assuming that we only use uppercase Latin letters) according to the suffixes of strings in S. The common prefix of two suffixes is shared. Each vertex has two boolean flags. The first/second one is to indicate that there exists a suffix/word in S *terminating* in that vertex, respectively. Example: If we have $S =$ {'CAR', 'CAT', 'RAT'}, we have the following suffixes {'CAR', 'AR', 'R', 'CAT', 'AT', 'T', 'RAT', 'AT', 'T'}. After sorting and removing duplicates, we have: {'AR', 'AT', 'CAR', 'CAT', 'R', 'RAT', 'T'}. Figure 6.2 shows the Suffix Trie with 7 suffix terminating vertices (filled circles) and 3 word terminating vertices (filled circles indicated with label 'In Dictionary').

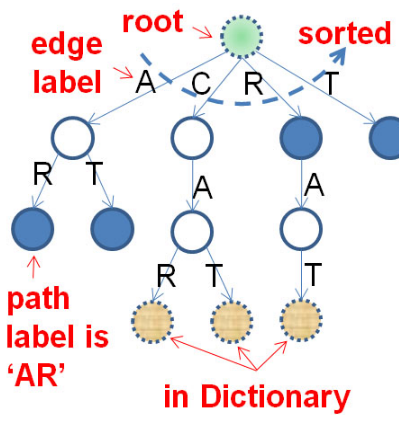


Figure 6.2: Suffix Trie

Suffix Trie is typically used as an efficient data structure for *dictionary*. Assuming that the Suffix Trie of a set of strings in the dictionary has been built, we can determine if a query/pattern string P exists in this dictionary (Suffix Trie) in $O(m)$ where m is the length of string P—this is efficient[10]. We do this by traversing the Suffix Trie from the root. For example, if we want to find if the word P = 'CAT' exists in the Suffix Trie shown in Figure 6.2, we can start from the root node, follow the edge with label 'C', then 'A', then 'T'. Since the vertex at this point has the word-terminating flag set to true, then we know that there is a word 'CAT' in the dictionary. Whereas, if we search for P = 'CAD', we go through this path: root → 'C' → 'A' but then we do not have edge with edge label 'D', so we conclude that 'CAD' is not in the dictionary.

Exercise 6.6.1.1*: Implement this Suffix Trie data structure using the ideas outlined above, i.e. create a vertex object with up to 26 ordered edges that represent 'A' to 'Z' and suffix/word terminating flags. Insert each suffix of each string in S into the Suffix Trie one by one. Analyze the time complexity of such Suffix Trie construction strategy and compare with Suffix Array construction strategy in Section 6.6.4! Also perform $O(m)$ queries for various pattern strings P by starting from the root and follow the corresponding edge labels.

---

[9] This is not a typo. The word 'TRIE' comes from the word 'information reTRIEval'.

[10] Another data structure for dictionary is balanced BST—see Section 2.3. It has $O(\log n \times m)$ performance for each dictionary query where n is the number of words in the dictionary. This is because one string comparison already costs $O(m)$.

6.6.2 Suffix Tree

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

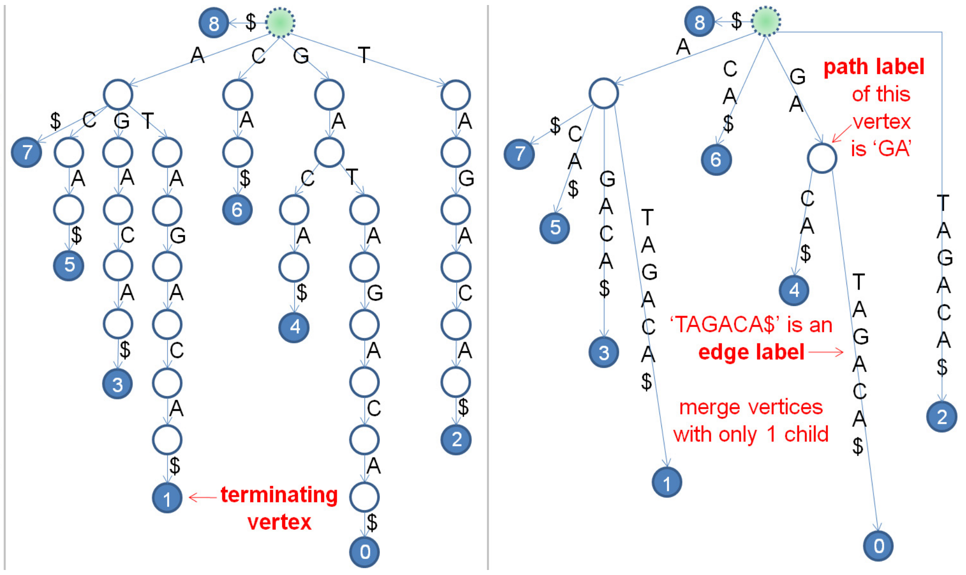

terminating vertex
path label of this vertex is 'GA'
'TAGACA$' is an edge label
merge vertices with only 1 child

Figure 6.3: Suffixes, Suffix Trie, and Suffix Tree of `T = 'GATAGACA$'`

Now, instead of working with several short strings, we work with one *long(er)* string. Consider a string `T = 'GATAGACA$'`. The last character `'$'` is a special terminating character appended to the original string `'GATAGACA'`. It has ASCII value lesser than the characters in `T`. This terminating character ensures that all suffixes terminate in leaf vertices.

The Suffix **Trie** of `T` is shown in Figure 6.3—middle. This time, the **terminating vertex** stores the *index* of the suffix that terminates in that vertex. Observe that the longer the string `T` is, there will be more duplicated vertices in the Suffix Trie. This can be inefficient. Suffix **Tree** of `T` is a Suffix Trie where we *merge* vertices with only one child (essentially a path compression). Compare Figure 6.3—middle and right to see this path compression process. Notice the **edge label** and **path label** in the figure. This time, the edge label can have more than one character. Suffix **Tree** is much more *compact* than Suffix **Trie** with at most $2n$ vertices only[11] (and thus at most $2n-1$ edges). Thus, rather than using Suffix Trie, we will use Suffix Tree in the subsequent sections.

Suffix Tree can be a new data structure for most readers of this book. Therefore in the third edition of this book, we have added a Suffix Tree visualization tool to show the structure of the Suffix Tree of any (but relatively short) input string `T` specified by the reader themselves. Several Suffix Tree applications shown in the next Section 6.6.3 are also included in the visualization.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/suffixtree.html`

Exercise 6.6.2.1*: Draw the Suffix Trie and the Suffix Tree of `T = 'COMPETITIVE$'`!
Hint: Use the Suffix Tree visualization tool shown above.

Exercise 6.6.2.2*: Given two vertices that represents two different suffixes, e.g. suffix 1 and suffix 5 in Figure 6.3—right, determine their Longest Common Prefix! (which is `'A'`).

---

[11]There are at most n leaves for n suffixes. All internal non-root vertices are always branching thus there can be at most $n-1$ such vertices. Total: n (leaves) + $(n-1)$ (internal nodes) + 1 (root) = $2n$ vertices.

6.6.3 Applications of Suffix Tree

Assuming that the Suffix Tree of a string `T` is *already built*, we can use it for these applications (not exhaustive):

String Matching in $O(m + occ)$

With Suffix Tree, we can find all (exact) occurrences of a pattern string `P` in `T` in $O(m+occ)$ where m is the length of the pattern string `P` itself and occ is the total number of occurrences of `P` in `T`—*no matter how long* the string `T` is. When the Suffix Tree is *already built*, this approach is *much faster* than string matching algorithms discussed earlier in Section 6.4.

Given the Suffix Tree of `T`, our task is to search for the vertex x in the Suffix Tree whose path label represents the pattern string `P`. Remember, a matching is after all a *common prefix* between pattern string `P` and some suffixes of string `T`. This is done by just one root to leaf traversal of Suffix Tree of `T` following the edge labels. Vertex with path label equals to `P` is the desired vertex x. Then, the suffix indices stored in the terminating vertices (leaves) of the subtree rooted at x are the occurrences of `P` in `T`.

Example: In the Suffix Tree of `T = 'GATAGACA$'` shown in Figure 6.4 and `P = 'A'`, we can simply traverse from root, go along the edge with edge label 'A' to find vertex x with the path label 'A'. There are 4 occurrences[12] of 'A' in the subtree rooted at x. They are suffix 7: `'A$'`, suffix 5: `'ACA$'`, suffix 3: `'AGACA$'`, and suffix 1: `'ATAGACA$'`.

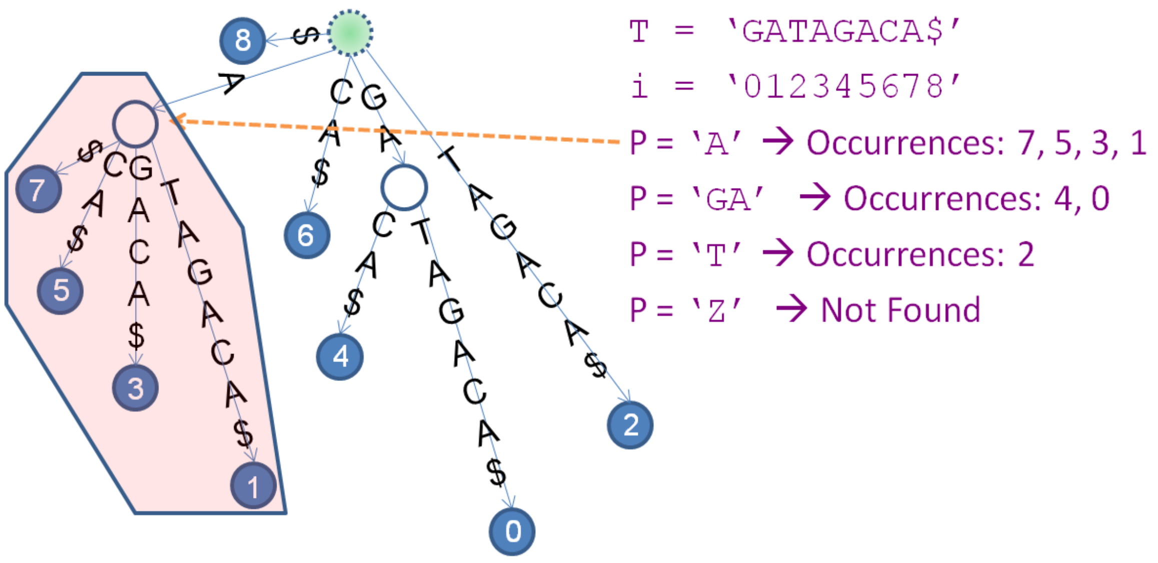


Figure 6.4: String Matching of `T = 'GATAGACA$'` with Various Pattern Strings

Finding the Longest Repeated Substring in $O(n)$

Given the Suffix Tree of `T`, we can also find the Longest Repeated Substring[13] (LRS) in T efficiently. The LRS problem is the problem of finding the longest substring of a string that occurs *at least twice*. The path label of the *deepest internal* vertex x in the Suffix Tree of T is the answer. Vertex x can be found with an $O(n)$ tree traversal. The fact that x is

[12]To be precise, occ is the *size* of subtree rooted at x, which can be larger—but not more than double—than the actual number (occ) of terminating vertices (leaves) in the subtree rooted at x.

[13]This problem has several interesting applications: Finding the chorus section of a song (that is repeated several times); Finding the (longest) repeated sentences in a (long) political speech, etc.

an internal vertex implies that it represents more than one suffixes of T (there will be > 1 terminating vertices in the subtree rooted at x) and these suffixes share a common prefix (which implies a repeated substring). The fact that x is the *deepest* internal vertex (from root) implies that its path label is the *longest* repeated substring.

Example: In the Suffix Tree of `T = 'GATAGACA$'` in Figure 6.5, the LRS is `'GA'` as it is the path label of the deepest internal vertex x—`'GA'` is repeated twice in `'GATAGACA$'`.

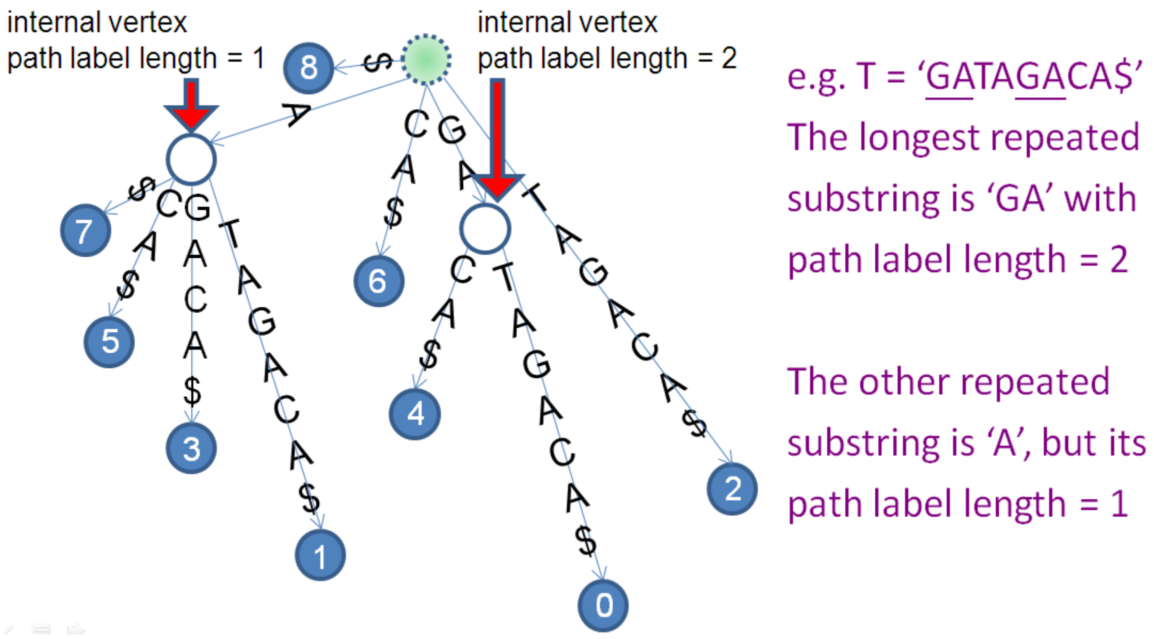

internal vertex
path label length = 1
internal vertex
path label length = 2
e.g. T = 'GATAGACA$'
The longest repeated substring is 'GA' with path label length = 2
The other repeated substring is 'A', but its path label length = 1

Figure 6.5: Longest Repeated Substring of `T = 'GATAGACA$'`

Finding the Longest Common Substring in $O(n)$

The problem of finding the Longest Common **Substring** (LCS[14]) of two **or more** strings can be solved in linear time[15] with Suffix Tree. Without loss of generality, let's consider the case with *two* strings only: T_1 and T_2. We can build a **generalized Suffix Tree** that combines the Suffix Tree of T_1 and T_2. To differentiate the source of each suffix, we use two different terminating vertex symbols, one for each string. Then, we mark *internal vertices* which have vertices in their subtrees with *different* terminating symbols. The suffixes represented by these marked internal vertices share a common prefix and come from *both* T_1 and T_2. That is, these marked internal vertices represent the common substrings between T_1 and T_2. As we are interested with the *longest* common substring, we report the path label of the *deepest* marked vertex as the answer.

For example, with T_1 = `'GATAGACA$'` and T_2 = `'CATA#'`, The Longest Common Substring is `'ATA'` of length 3. In Figure 6.6, we see the vertices with path labels `'A'`, `'ATA'`, `'CA'`, and `'TA'` have two different terminating symbols (notice that vertex with path label `'GA'` is *not* considered as both suffix `'GACA$'` and `'GATAGACA$'` come from T_1). These are the common substrings between T_1 and T_2. The deepest marked vertex is `'ATA'` and this is the longest common substring between T_1 and T_2.

[14]Note that 'Substring' is different from 'Subsequence'. For example, "BCE" is a subsequence but not a substring of "ABCDEF" whereas "BCD" (contiguous) is both a subsequence and a substring of "ABCDEF".

[15]Only if we use the linear time Suffix Tree construction algorithm (not discussed in this book, see [65]).

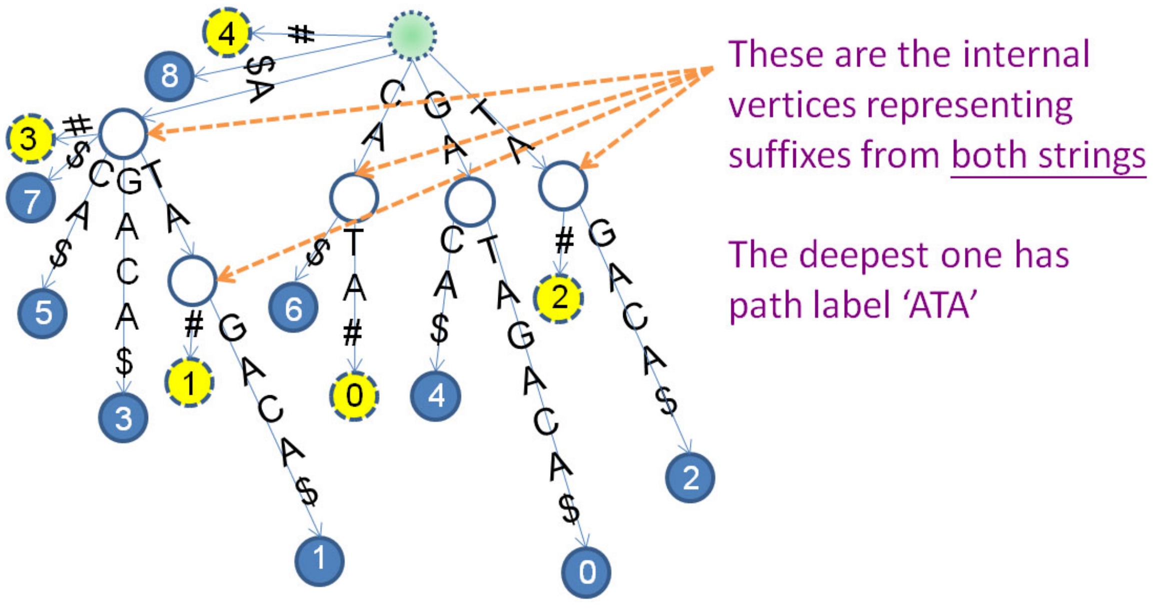

These are the internal vertices representing suffixes from both strings
The deepest one has path label 'ATA'

Figure 6.6: Generalized ST of T_1 = 'G<u>ATA</u>GACA$' and T_2 = 'C<u>ATA</u>#' and their LCS

Exercise 6.6.3.1: Given the same Suffix Tree in Figure 6.4, find `P = 'CA'` and `P = 'CAT'`!

Exercise 6.6.3.2: Find the LRS in `T = 'CGACATTACATTA$'`! Build the Suffix Tree first.

Exercise 6.6.3.3*: Instead of finding the LRS, we now want to find the repeated substring *that occurs the most*. Among several possible candidates, pick the longest one. For example, if `T = 'DEFG1ABC2DEFG3ABC4ABC$'`, the answer is `'ABC'` of length 3 that occurs three times (not `'BC'` of length 2 or `'C'` of length 1 which also occur three times) instead of `'DEFG'` of length 4 that occurs only two times. Outline the strategy to find the solution!

Exercise 6.6.3.4: Find the LCS of T_1 = `'STEVEN$'` and T_2 = `'SEVEN#'`!

Exercise 6.6.3.5*: Think of how to generalize this approach to find the LCS of *more than two strings*. For example, given three strings T_1 = `'STEVEN$'`, T_2 = `'SEVEN#'`, and T_3 = `'EVE@'`, how to determine that their LCS is `'EVE'`?

Exercise 6.6.3.6*: Customize the solution further so that we find the LCS of k *out of* n *strings*, where $k \leq n$. For example, given the same three strings T_1, T_2, and T_3 as above, how to determine that the LCS of 2 out of 3 strings is `'EVEN'`?

6.6.4 Suffix Array

In the previous subsection, we have shown several string processing problems that can be solved *if the Suffix Tree is already built*. However, the efficient implementation of linear time Suffix Tree construction (see [65]) is complex and thus risky under programming contest setting. Fortunately, the next data structure that we are going to describe—the **Suffix Array** invented by Udi Manber and Gene Myers [43]—has similar functionalities as Suffix Tree but (much) simpler to construct and use, especially in programming contest setting. Thus, we will skip the discussion on $O(n)$ Suffix Tree construction [65] and instead focus on the $O(n \log n)$ Suffix Array construction [68] which is easier to use. Then, in the next subsection, we will show that we can apply Suffix Array to solve problems that have been shown to be solvable with Suffix Tree.

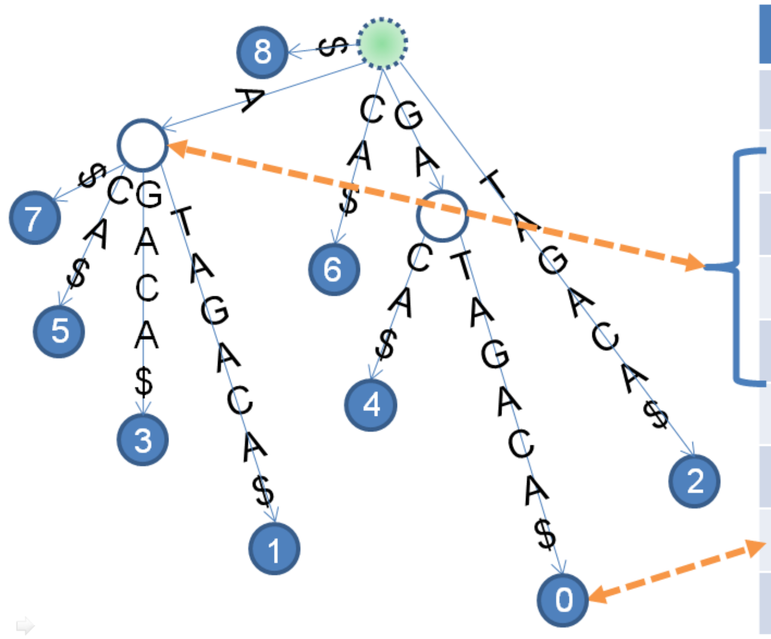
i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

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$

Figure 6.7: Sorting the Suffixes of `T = 'GATAGACA$'`

Basically, Suffix Array is an integer array that stores a permutation of n indices of *sorted* suffixes. For example, consider the same `T = 'GATAGACA$'` with $n = 9$. The Suffix Array of `T` is a permutation of integers `[0..n-1]` = {`8, 7, 5, 3, 1, 6, 4, 0, 2`} as shown in Figure 6.7. That is, the suffixes in sorted order are suffix `SA[0]` = suffix `8` = '`$`', suffix `SA[1]` = suffix `7` = '`A$`', suffix `SA[2]` = suffix `5` = '`ACA$`', ..., and finally suffix `SA[8]` = suffix 2 = '`TAGACA$`'.



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$

Figure 6.8: Suffix Tree and Suffix Array of `T = 'GATAGACA$'`

Suffix Tree and Suffix Array are closely related. As we can see in Figure 6.8, the tree traversal of the Suffix Tree visits the terminating vertices (the leaves) in Suffix Array order. An **internal vertex** in Suffix Tree corresponds to a **range** in Suffix Array (a collection of sorted suffixes that share a common prefix). A **terminating vertex** (always at leaf due to the usage of a terminating character) in Suffix Tree corresponds to an **individual index** in Suffix Array (a single suffix). Keep these similarities in mind. They will be useful in the next subsection when we discuss applications of Suffix Array.

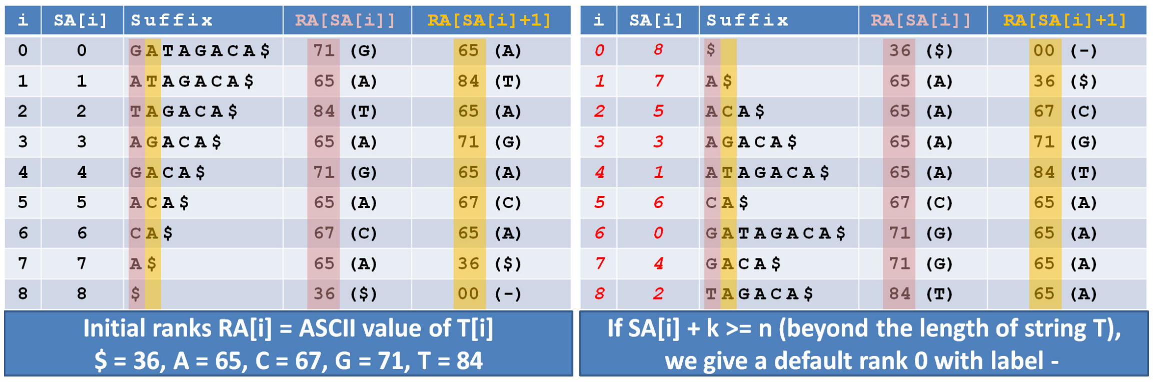
Suffix Array is good enough for many challenging string problems involving *long strings* in programming contests. Here, we present two ways to construct a Suffix Array given a string `T[0..n-1]`. The first one is very simple, as shown below:

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

#define MAX_N 1010                          // first approach: O(n^2 log n)
char T[MAX_N];    // this naive SA construction cannot go beyond 1000 chars
int SA[MAX_N], i, n;                         // in programming contest settings

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

int main() {
  n = (int)strlen(gets(T)); // read line and immediately compute its length
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp);     // sort: O(n log n) * cmp: O(n) = O(n^2 log n)
  for (i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

When applied to string `T = 'GATAGACA$'`, the simple code above that sorts all suffixes with built-in sorting and string comparison *library* produces the correct Suffix Array = `{8, 7, 5, 3, 1, 6, 4, 0, 2}`. However, this is barely useful except for contest problems with $n \leq 1000$. The overall runtime of this algorithm is $O(n^2 \log n)$ because the `strcmp` operation that is used to determine the order of two (possibly long) suffixes is too costly, up to $O(n)$ per one pair of suffix comparison.

A *better way* to construct Suffix Array is to sort the *ranking pairs* (small integers) of suffixes in $O(\log_2 n)$ iterations from $k = 1, 2, 4, \ldots,$ the last **power of 2** that is less than n. At each iteration, this construction algorithm sorts the suffixes based on the ranking pair `(RA[SA[i]], RA[SA[i]+k])` of suffix `SA[i]`. This algorithm is based on the discussion in [68]. An example execution is shown below for `T = 'GATAGACA$'` and $n = 9$.

- First, `SA[i] = i` and `RA[i] =` ASCII value of `T[i]` $\forall$`i` $\in$ `[0..n-1]` (Table 6.1—left). At iteration $k = 1$, the ranking pair of suffix `SA[i]` is `(RA[SA[i]], RA[SA[i]+1])`.



i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	0	GATAGACA$	71 (G)	65 (A)
1	1	ATAGACA$	65 (A)	84 (T)
2	2	TAGACA$	84 (T)	65 (A)
3	3	AGACA$	65 (A)	71 (G)
4	4	GACA$	71 (G)	65 (A)
5	5	ACA$	65 (A)	67 (C)
6	6	CA$	67 (C)	65 (A)
7	7	A$	65 (A)	36 ($)
8	8	$	36 ($)	00 (-)

Initial ranks RA[i] = ASCII value of T[i]
$ = 36, A = 65, C = 67, G = 71, T = 84

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	8	$	36 ($)	00 (-)
1	7	A$	65 (A)	36 ($)
2	5	ACA$	65 (A)	67 (C)
3	3	AGACA$	65 (A)	71 (G)
4	1	ATAGACA$	65 (A)	84 (T)
5	6	CA$	67 (C)	65 (A)
6	0	GATAGACA$	71 (G)	65 (A)
7	4	GACA$	71 (G)	65 (A)
8	2	TAGACA$	84 (T)	65 (A)

If SA[i] + k >= n (beyond the length of string T), we give a default rank 0 with label -

Table 6.1: L/R: Before/After Sorting; k = 1; the initial sorted order appears

Example 1: The rank of suffix 5 ‘ACA$’ is (‘A’, ‘C’) = (65, 67).

Example 2: The rank of suffix 3 ‘AGACA$’ is (‘A’, ‘G’) = (65, 71).

After we sort these ranking pairs, the order of suffixes is now like Table 6.1—right, where suffix 5 ‘ACA$’ comes before suffix 3 ‘AGACA$’, etc.

- At iteration $k = 2$, the ranking pair of suffix SA[i] is (RA[SA[i]], RA[SA[i]+2]). This ranking pair is now obtained by looking at the first pair and the second pair of characters only. To get the new ranking pairs, we do not have to recompute many things. We set the first one, i.e. Suffix 8 ‘$’ to have new rank $r = 0$. Then, we iterate from i = [1..n-1]. If the ranking pair of suffix SA[i] is different from the ranking pair of the previous suffix SA[i-1] in sorted order, we increase the rank $r = r + 1$. Otherwise, the rank stays at r (see Table 6.2—left).

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	0	GATAGACA$	6 (GA)	7 (TA)
7	4	GACA$	6 (GA)	5 (CA)
8	2	TAGACA$	7 (TA)	6 (GA)

$- (first item) is given rank 0, then for i = 1 to n-1, compare rank pair of this row with previous row

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	4	GACA$	6 (GA)	5 (CA)
7	0	GATAGACA$	6 (GA)	7 (TA)
8	2	TAGACA$	7 (TA)	6 (GA)

If SA[i] + k >= n (beyond the length of string T), we give a default rank 0 with label -

Table 6.2: L/R: Before/After Sorting; k = 2; ‘GATAGACA’ and ‘GACA’ are swapped

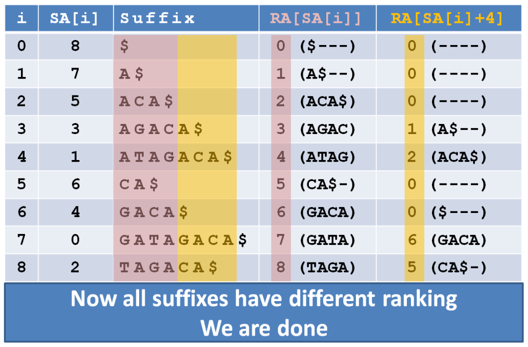
Example 1: In Table 6.1—right, the ranking pair of suffix 7 ‘A$’ is (65, 36) which is different with the ranking pair of previous suffix 8 ‘$-’ which is (36, 0). Therefore in Table 6.2—left, suffix 7 has a new rank 1.

Example 2: In Table 6.1—right, the ranking pair of suffix 4 ‘GACA$’ is (71, 65) which is similar with the ranking pair of previous suffix 0 ‘GATAGACA$’ which is also (71, 65). Therefore in Table 6.2—left, since suffix 0 is given a new rank 6, then suffix 4 is also given the same new rank 6.

Once we have updated RA[SA[i]] ∀i ∈ [0..n-1], the value of RA[SA[i]+k] can be easily determined too. In our explanation, if SA[i]+k $\geq$ n, we give a default rank 0. See **Exercise 6.6.4.2*** for more details on the implementation aspect of this step.

At this stage, the ranking pair of suffix 0 ‘GATAGACA$’ is (6, 7) and suffix 4 ‘GACA$’ is (6, 5). These two suffixes are still not in sorted order whereas all the other suffixes are already in their correct order. After another round of sorting, the order of suffixes is now like Table 6.2—right.

- At iteration $k = 4$, the ranking pair of suffix SA[i] is (RA[SA[i]], RA[SA[i]+4]). This ranking pair is now obtained by looking at the first quadruple and the second quadruple of characters only. Now, notice that the previous ranking pairs of Suffix 4 (6, 5) and Suffix 0 (6, 7) in Table 6.2—right are now different. Therefore, after re-ranking, all n suffixes in Table 6.3 now have different ranking. This can be easily verified by checking if RA[SA[n-1]] == n-1. When this happens, we have successfully obtained the Suffix Array. Notice that the major sorting work is done in the first few iterations only and we usually do not need many iterations.



i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+4]
0	8	$	0 ($---)	0 (----)
1	7	A$	1 (A$--)	0 (----)
2	5	ACA$	2 (ACA$)	0 (----)
3	3	AGACA$	3 (AGAC)	1 (A$--)
4	1	ATAGACA$	4 (ATAG)	2 (ACA$)
5	6	CA$	5 (CA$-)	0 (----)
6	4	GACA$	6 (GACA)	0 ($---)
7	0	GATAGACA$	7 (GATA)	6 (GACA)
8	2	TAGACA$	8 (TAGA)	5 (CA$-)

Now all suffixes have different ranking
We are done

Table 6.3: Before/After sorting; k = 4; no change

This Suffix Array construction algorithm can be new for most readers of this book. Therefore in the third edition of this book, we have added a Suffix Array visualization tool to show the steps of of any (but relatively short) input string `T` specified by the reader themselves. Several Suffix Array applications shown in the next Section 6.6.5 are also included.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/suffixarray.html`

We can implement the sorting of ranking pairs above using (built-in) $O(n \log n)$ sorting library. As we repeat the sorting process up to $\log n$ times, the overall time complexity is $O(\log n \times n \log n) = O(n \log^2 n)$. With this time complexity, we can now work with strings of length up to $\approx 10K$. However, since the sorting process only sort *pair of small integers*, we can use a *linear time* two-pass Radix Sort (that internally calls Counting Sort—see more details in Section 9.32) to reduce the sorting time to $O(n)$. As we repeat the sorting process up to $\log n$ times, the overall time complexity is $O(\log n \times n) = O(n \log n)$. Now, we can work with strings of length up to $\approx 100K$—typical programming contest range.

We provide our $O(n \log n)$ implementation below. Please scrutinize the code to understand how it works. For ICPC contestants only: As you can bring hard copy materials to the contest, it is a good idea to put this code in your team's library.

```
#define MAX_N 100010                          // second approach: O(n log n)
char T[MAX_N];                        // the input string, up to 100K characters
int n;                                          // the length of input string
int RA[MAX_N], tempRA[MAX_N];           // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];       // suffix array and temporary suffix array
int c[MAX_N];                                      // for counting/radix sort

void countingSort(int k) {                                              // O(n)
  int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                             // clear frequency table
  for (i = 0; i < n; i++)         // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t; }
  for (i = 0; i < n; i++)            // shuffle the suffix array if necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                         // update the suffix array SA
    SA[i] = tempSA[i];
}
```

```
void constructSA() {         // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                 // initial rankings
  for (i = 0; i < n; i++) SA[i] = i;     // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {       // repeat sorting process log n times
    countingSort(k);  // actually radix sort: sort based on the second item
    countingSort(0);          // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                    // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                     // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization trick
} }

int main() {
  n = (int)strlen(gets(T));       // input T as per normal, without the '$'
  T[n++] = '$';                              // add terminating character
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

Exercise 6.6.4.1*: Show the steps to compute the Suffix Array of `T = 'COMPETITIVE$'` with $n = 12$! How many sorting iterations that you need to get the Suffix Array?
Hint: Use the Suffix Array visualization tool shown above.

Exercise 6.6.4.2*: In the suffix array code shown above, will the following line:

```
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
```

causes index out of bound in some cases?
That is, will `SA[i]+k` or `SA[i-1]+k` ever be $\geq$ `n` and crash the program? Explain!

Exercise 6.6.4.3*: Will the suffix array code shown above works if the input string `T` contains a space (ASCII value = 32) inside? Hint: The default terminating character used—i.e. `'$'`—has ASCII value = 36.

6.6.5 Applications of Suffix Array

We have mentioned earlier that Suffix Array is closely related to Suffix Tree. In this subsection, we show that with Suffix Array (which is easier to construct), we can solve the string processing problems shown in Section 6.6.3 that are solvable using Suffix Tree.

String Matching in $O(m \log n)$

After we obtain the Suffix Array of `T`, we can search for a pattern string `P` (of length m) in `T` (of length n) in $O(m \log n)$. This is a factor of $\log n$ times slower than the Suffix Tree version but in practice is quite acceptable. The $O(m \log n)$ complexity comes from the fact that we can do two $O(\log n)$ binary searches on sorted suffixes and do up to $O(m)$ suffix comparisons[16]. The first/second binary search is to find the lower/upper bound respectively. This lower/upper bound is the the smallest/largest `i` such that the prefix of suffix `SA[i]` matches the pattern string `P`, respectively. All the suffixes between the lower and upper bound are the occurrences of pattern string `P` in `T`. Our implementation is shown below:

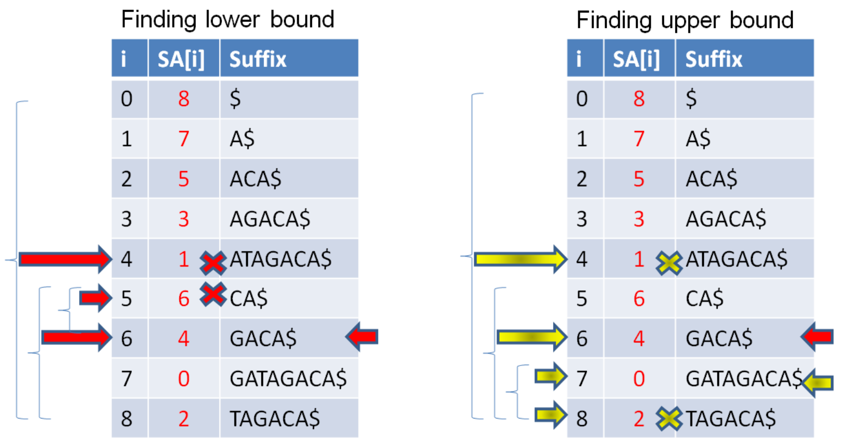
```
ii stringMatching() {                          // string matching in O(m log n)
  int lo = 0, hi = n-1, mid = lo;                 // valid matching = [0..n-1]
  while (lo < hi) {                                          // find lower bound
    mid = (lo + hi) / 2;                                    // this is round down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix 'mid'
    if (res >= 0) hi = mid;        // prune upper half (notice the >= sign)
    else          lo = mid + 1;           // prune lower half including mid
  }                                           // observe '=' in "res >= 0" above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {            // if lower bound is found, find upper bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                                   // prune upper half
    else         lo = mid + 1;            // prune lower half including mid
  }                           // (notice the selected branch when res == 0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;                 // special case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second item of the pair, respectively

int main() {
  n = (int)strlen(gets(T));       // input T as per normal, without the '$'
  T[n++] = '$';                                    // add terminating character
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);

  while (m = (int)strlen(gets(P)), m) {     // stop if P is an empty string
    ii pos = stringMatching();
    if (pos.first != -1 && pos.second != -1) {
      printf("%s found, SA [%d..%d] of %s\n", P, pos.first, pos.second, T);
      printf("They are:\n");
      for (int i = pos.first; i <= pos.second; i++)
        printf("  %s\n", T + SA[i]);
    } else printf("%s is not found in %s\n", P, T);
} } // return 0;
```

A sample execution of this string matching algorithm on the Suffix Array of `T = 'GATAGACA$'` with `P = 'GA'` is shown in Table 6.4 below.

We start by finding the lower bound. The current range is `i = [0..8]` and thus the middle one is `i = 4`. We compare the first two characters of suffix `SA[4]`, which is `'ATAGACA$'`, with `P = 'GA'`. As `P = 'GA'` is larger, we continue exploring `i = [5..8]`. Next, we compare the first two characters of suffix `SA[6]`, which is `'GACA$'`, with `P = 'GA'`. It is a match.

[16]This is achievable by using the `strncmp` function to compare only the first m characters of both suffixes.

As we are currently looking for the *lower* bound, we do not stop here but continue exploring `i = [5..6]`. `P = 'GA'` is larger than suffix `SA[5]`, which is `'CA$'`. We stop here. Index `i = 6` is the lower bound, i.e. suffix `SA[6]`, which is `'GACA$'`, is the *first* time pattern `P = 'GA'` appears as a prefix of a suffix in the list of sorted suffixes.



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$

Table 6.4: String Matching using Suffix Array

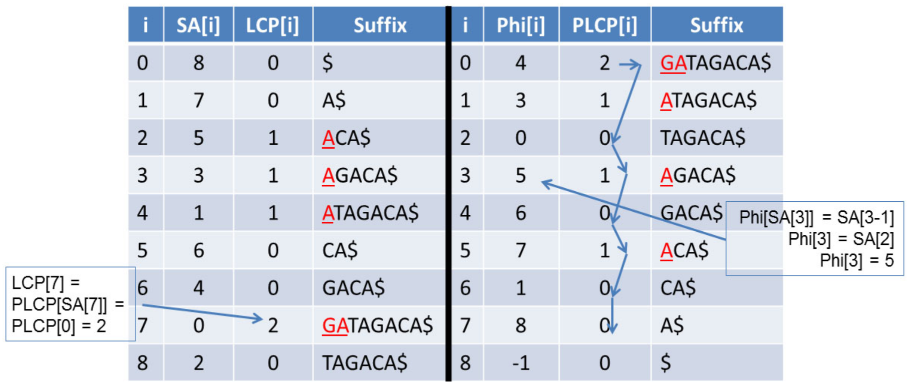
Next, we search for the upper bound. The first step is the same as above. But at the second step, we have a match between suffix `SA[6]`, which is `'GACA$'`, with `P = 'GA'`. Since now we are looking for the *upper* bound, we continue exploring `i = [7..8]`. We find another match when comparing suffix `SA[7]`, which is `'GATAGACA$'`, with `P = 'GA'`. We stop here. This `i = 7` is the upper bound in this example, i.e. suffix `SA[7]`, which is `'GATAGACA$'`, is the *last* time pattern `P = 'GA'` appears as a prefix of a suffix in the list of sorted suffixes.

Finding the Longest Common Prefix in $O(n)$

Given the Suffix Array of `T`, we can compute the Longest Common Prefix (LCP) between *consecutive* suffixes in Suffix Array order. By definition, `LCP[0] = 0` as suffix `SA[0]` is the first suffix in Suffix Array order without any other suffix preceding it. For `i > 0, LCP[i]` = the length of common prefix between suffix `SA[i]` and suffix `SA[i-1]`. See Table 6.5—left. We can compute LCP directly by definition by using the code below. However, this approach is slow as it can increase the value of L up to $O(n^2)$ times. This defeats the purpose of building Suffix Array in $O(n \log n)$ time as shown in Section 6.8.

```
void computeLCP_slow() {
  LCP[0] = 0;                                          // default value
  for (int i = 1; i < n; i++) {                  // compute LCP by definition
    int L = 0;                                        // always reset L to 0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;       // same L-th char, L++
    LCP[i] = L;
} }
```

A better solution using the Permuted Longest-Common-Prefix (PLCP) theorem [37] is described below. The idea is simple: It is *easier* to compute the LCP in the original position order of the suffixes instead of the lexicographic order of the suffixes. In Table 6.5—right, we have the original position order of the suffixes of `T = 'GATAGACA$'`. Observe that column `PLCP[i]` forms a pattern: Decrease-by-1 block ($2 \to 1 \to 0$); increase to 1; decrease-by-1 block again ($1 \to 0$); increase to 1 again; decrease-by-1 block again ($1 \to 0$), etc.



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

LCP[7] =
PLCP[SA[7]] =
PLCP[0] = 2

Phi[SA[3]] = SA[3-1]
Phi[3] = SA[2]
Phi[3] = 5

Table 6.5: Computing the LCP given the SA of `T = 'GATAGACA$'`

The PLCP theorem says that the total number of increase (and decrease) operations is at most $O(n)$. This pattern and this $O(n)$ guarantee are exploited in the code below.

First, we compute `Phi[i]`, that stores the suffix index of the previous suffix of suffix `SA[i]` in Suffix Array order. By definition, `Phi[SA[0]] = -1`, i.e. there is no previous suffix that precede suffix `SA[0]`. Take some time to verify the correctness of column `Phi[i]` in Table 6.5—right. For example, `Phi[SA[3]] = SA[3-1]`, so `Phi[3] = SA[2] = 5`.

Now, with `Phi[i]`, we can compute the permuted LCP. The first few steps of this algorithm is elaborated below. When `i = 0`, we have `Phi[0] = 4`. This means suffix 0 `'GATAGACA$'` has suffix 4 `'GACA$'` before it in Suffix Array order. The first two characters (`L = 2`) of these two suffixes match, so `PLCP[0] = 2`.

When `i = 1`, we know that *at least* `L-1 = 1` characters can match as the next suffix in position order will have one less starting character than the current suffix. We have `Phi[1] = 3`. This means suffix 1 `'ATAGACA$'` has suffix 3 `'AGACA$'` before it in Suffix Array order. Observe that these two suffixes indeed have at least 1 character match (that is, we do not start from `L = 0` as in `computeLCP_slow()` function shown earlier and therefore this is more efficient). As we cannot extend this further, we have `PLCP[1] = 1`.

We continue this process until `i = n-1`, bypassing the case when `Phi[i] = -1`. As the PLCP theorem says that L will be increased/decreased at most n times, this part runs in amortized $O(n)$. Finally, once we have the PLCP array, we can put the permuted LCP back to the correct position. The code is relatively short, as shown below.

```
void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                             // default value
  for (i = 1; i < n; i++)                                   // compute Phi in O(n)
    Phi[SA[i]] = SA[i-1];    // remember which suffix is behind this suffix
  for (i = L = 0; i < n; i++) {                   // compute Permuted LCP in O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }               // special case
    while (T[i + L] == T[Phi[i] + L]) L++;        // L increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                              // L decreased max n times
  }
  for (i = 0; i < n; i++)                                   // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];    // put the permuted LCP to the correct position
}
```

Finding the Longest Repeated Substring in $O(n)$

If we have computed the Suffix Array in $O(n \log n)$ and the LCP between consecutive suffixes in Suffix Array order in $O(n)$, then we can determine the length of the Longest Repeated Substring (LRS) of `T` in $O(n)$.

The length of the longest repeated substring is just the highest number in the LCP array. In Table 6.5—left that corresponds to the Suffix Array and the LCP of `T = 'GATAGACA$'`, the highest number is 2 at index `i = 7`. The first 2 characters of the corresponding suffix `SA[7]` (suffix `0`) is `'GA'`. This is the longest repeated substring in `T`.

Finding the Longest Common Substring in $O(n)$

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#

Table 6.6: The Suffix Array, LCP, and owner of `T = 'GATAGACA$CATA#'`

Without loss of generality, let's consider the case with only *two* strings. We use the same example as in the Suffix Tree section earlier: T_1 `= 'GATAGACA$'` and T_2 `= 'CATA#'`. To solve the LCS problem using Suffix Array, first we have to concatenate both strings (note that the terminating characters of both strings *must be different*) to produce `T = 'GATAGACA$CATA#'`. Then, we compute the Suffix and LCP array of `T` as shown in Figure 6.6.

Then, we go through consecutive suffixes in $O(n)$. If two consecutive suffixes belong to different owner (can be easily checked[17], for example we can test if suffix `SA[i]` belongs to T_1 by testing if `SA[i]` $<$ the length of T_1), we look at the LCP array and see if the maximum LCP found so far can be increased. After one $O(n)$ pass, we will be able to determine the Longest Common Substring. In Figure 6.6, this happens when `i = 7`, as suffix `SA[7]` $=$ suffix `1` $=$ `'ATAGACA$CATA#'` (owned by T_1) and its previous suffix `SA[6]` $=$ suffix `10` $=$ `'ATA#'` (owned by T_2) have a common prefix of length 3 which is `'ATA'`. This is the LCS.

We close this section and this chapter by highlighting the availability of our source code. Please spend some time understanding the source code which may not be trivial for those who are new with Suffix Array.

Source code: `ch6_04_sa.cpp/java`

[17]With three or more strings, this check will have more 'if statements'.

Exercise 6.6.5.1*: Suggest some possible improvements to the `stringMatching()` function shown in this section!

Exercise 6.6.5.2*: Compare the KMP algorithm shown in Section 6.4 with the string matching feature of Suffix Array. When it is more beneficial to use Suffix Array to deal with string matching and when it is more beneficial to just use KMP or standard string libraries?

Exercise 6.6.5.3*: Solve all exercises on Suffix Tree applications, i.e. **Exercise 6.6.3.1, 2, 3*, 4, 5*, and 6*** using Suffix Array instead!

Programming Exercises related to Suffix Array[18]:

1. UVa 00719 - Glass Beads (min lexicographic rotation[19]; $O(n \log n)$ build SA)
2. **UVa 00760 - DNA Sequencing *** (Longest Common *Substring* of two strings)
3. UVa 01223 - Editor (LA 3901, Seoul07, Longest Repeated Substring (or KMP))
4. *UVa 01254 - Top 10* (LA 4657, Jakarta09, Suffix Array + Segment Tree)
5. **UVa 11107 - Life Forms *** (Longest Common Substring of $> \frac{1}{2}$ of the strings)
6. **UVa 11512 - GATTACA *** (Longest Repeated Substring)
7. SPOJ 6409 - Suffix Array (problem author: Felix Halim)
8. IOI 2008 - Type Printer (DFS traversal of Suffix Trie)

[18]You can try solving these problems with Suffix Tree, but you have to learn how to code the Suffix Tree construction algorithm by yourself. The programming problems listed here are solvable with Suffix Array. Also please take note that our sample code uses `gets` for reading the input strings. If you use `scanf(‘‘%s’’)` or `getline`, do not forget to adjust the potential DOS/UNIX ‘end of line’ differences.

[19]This problem can be solved by concatenating the string with itself, build the Suffix Array, then find the first suffix in Suffix Array sorted order that has length greater or equal to n.

6.7 Solution to Non-Starred Exercises

C Solutions for Section 6.2

Exercise 6.2.1:

(a) A string is stored as an array of characters terminated by null, e.g. `char str[30x10+50], line[30+50];`. It is a good practice to declare array size slightly bigger than requirement to avoid "off by one" bug.

(b) To read the input line by line, we use[20] `gets(line);` or `fgets(line, 40, stdin);` in `string.h` (or `cstring`) library. Note that `scanf(‘‘%s’’, line)` is not suitable here as it will only read the first word.

(c) We first set `strcpy(str, ‘‘’’);`, and then we combine the `line`s that we read into a longer string using `strcat(str, line);`. If the current line is not the last one, we append a space to the back of `str` using `strcat(str, ‘‘ ’’);` so that the last word from this line is not accidentally combined with the first word of the next line.

(d) We stop reading the input when `strncmp(line, ‘‘.......’’, 7) == 0`. Note that `strncmp` only compares the first n characters.

Exercise 6.2.2:

(a) For finding a substring in a relatively short string (the standard string matching problem), we can just use library function. We can use `p = strstr(str, substr);` The value of `p` will be `NULL` if `substr` is not found in `str`.

(b) If there are multiple copies of `substr` in `str`, we can use `p = strstr(str + pos, substr)`. Initially `pos = 0`, i.e. we search from the first character of `str`. After finding one occurrence of `substr` in `str`, we can call `p = strstr(str + pos, substr)` again where this time `pos` is the index of the current occurrence of `substr` in `str` *plus one* so that we can get the next occurrence. We repeat this process until `p == NULL`. This C solution requires understanding of the memory address of a C array.

Exercise 6.2.3: In many string processing tasks, we are required to iterate through every characters in `str` once. If there are n characters in `str`, then such scan requires $O(n)$. In both C/C++, we can use `tolower(ch)` and `toupper(ch)` in `ctype.h` to convert a character to its lower and uppercase version. There are also `isalpha(ch)`/`isdigit(ch)` to check whether a given character is alphabet `[A-Za-z]`/digit, respectively. To test whether a character is a vowel, one method is to prepare a string `vowel = "aeiou";` and check if the given character is one of the five characters in `vowel`. To check whether a character is a consonant, simply check if it is an alphabet but not a vowel.

Exercise 6.2.4: Combined C and C++ solutions:

(a) One of the easiest ways to tokenize a string is to use `strtok(str, delimiters);` in C.

(b) These tokens can then be stored in a C++ `vector<string> tokens`.

(c) We can use C++ STL `algorithm::sort` to sort `vector<string> tokens`. When needed, we can convert C++ `string` back to C string by using `str.c_str()`.

[20]Note: Function `gets` is actually unsafe because it does not perform bounds checking on input size.

Exercise 6.2.5: See the C++ solution.

Exercise 6.2.6: Read the input character by character and count incrementally, look for the presence of '`\n`' that signals the end of a line. Pre-allocating a fixed-sized buffer is not a good idea as the problem author can set a ridiculously long string to break your code.

C++ Solutions for Section 6.2

Exercise 6.2.1:

(a) We can use class `string`.

(b) We can use `cin.getline()` in `string` library.

(c) We can use the '+' operator directly to concatenate strings.

(d) We can use the '==' operator directly to compare two strings.

Exercise 6.2.2:

(a) We can use function `find` in class `string`.

(b) Same idea as in C language. We can set the offset value in the second parameter of function `find` in class `string`.

Exercise 6.2.3-4: Same solutions as in C language.

Exercise 6.2.5: We can use C++ STL `map<string, int>` to keep track the frequency of each word. Every time we encounter a new token (which is a string), we increase the corresponding frequency of that token by one. Finally, we scan through all tokens and determine the one with the highest frequency.

Exercise 6.2.6: Same solution as in C language.

Java Solutions for Section 6.2

Exercise 6.2.1:

(a) We can use class `String`, `StringBuffer`, or `StringBuilder` (this one is faster than `StringBuffer`).

(b) We can use the `nextLine` method in Java `Scanner`. For faster I/O, we can consider using the `readLine` method in Java `BufferedReader`.

(c) We can use the `append` method in `StringBuilder`. We should not concatenate Java Strings with the '+' operator as Java String class is immutable and thus such operation is (very) costly.

(d) We can use the `equals` method in Java `String`.

Exercise 6.2.2:

(a) We can use the `indexOf` method in class `String`.

(b) Same idea as in C language. We can set the offset value in the second parameter of `indexOf` method in class `String`.

Exercise 6.2.3: Use Java `StringBuilder` and `Character` classes for these operations.

Exercise 6.2.4:

(a) We can use Java `StringTokenizer` class or the `split` method in Java `String` class.

(b) We can use Java `Vector` of `String`s.

(c) We can use Java `Collections.sort`.

Exercise 6.2.5: Same idea as in C++ language.
We can use Java `TreeMap<String, Integer>`.

Exercise 6.2.6: We need to use the `read` method in Java `BufferedReader` class.

Solutions for the Other Sections

Exercise 6.5.1.1: Different scoring scheme will yield different (global) alignment. If given a string alignment problem, read the problem statement and see what is the required cost for match, mismatch, insert, and delete. Adapt the algorithm accordingly.

Exercise 6.5.1.2: You have to save the predecessor information (the arrows) during the DP computation. Then follow the arrows using recursive backtracking. See Section 3.5.1.

Exercise 6.5.1.3: The DP solution only need to refer to previous row so it can utilize the 'space saving trick' by just using two rows, the current row and the previous row. The new space complexity is just $O(min(n, m))$, that is, put the string with the lesser length as string 2 so that each row has lesser columns (less memory). The time complexity of this solution is still $O(nm)$. The only drawback of this approach, as with any other space saving trick is that we will not be able to reconstruct the optimal solution. So if the actual optimal solution is needed, we cannot use this space saving trick. See Section 3.5.1.

Exercise 6.5.1.4: Simply concentrate along the main diagonal with width d. We can speed up Needleman-Wunsch's algorithm to $O(dn)$ by doing this.

Exercise 6.5.1.5: It involves Kadane's algorithm again (see maximum sum problem in Section 3.5.2).

Exercise 6.5.2.1: `'pple'`.

Exercise 6.5.2.2: Set score for match = 0, mismatch = 1, insert and delete = negative infinity. However, this solution is not efficient and not natural, as we can simply use an $O(min(n, m))$ algorithm to scan both string 1 and string 2 and count how many characters are different.

Exercise 6.5.2.3: Reduced to LIS, $O(n \log k)$ solution. The reduction to LIS is not shown. Draw it and see how to reduce this problem into LIS.

Exercise 6.6.3.1: 'CA' is found, 'CAT' is not.

Exercise 6.6.3.2: 'ACATTA'.

Exercise 6.6.3.4: 'EVEN'.

6.8 Chapter Notes

The material about String Alignment (Edit Distance), Longest Common Subsequence, and Suffix Trie/Tree/Array are originally from **A/P Sung Wing Kin, Ken** [62], School of Computing, National University of Singapore. The material have since evolved from a more theoretical style into the current competitive programming style.

The section about basic string processing skills (Section 6.2) and the Ad Hoc string processing problems were born from our experience with string-related problems and techniques. The number of programming exercises mentioned there is about three quarters of all other string processing problems discussed in this chapter. We are aware that these are not the typical ICPC problems/IOI tasks, but they are still good programming exercises to improve your programming skills.

In Section 6.4, we discuss the library solutions and one fast algorithm (Knuth-Morris-Pratt/KMP algorithm) for the String Matching problem. The KMP implementation will be useful if you have to modify basic string matching requirement yet you still need fast performance. We believe KMP is fast enough for finding pattern string in a long string for typical contest problems. Through experimentation, we conclude that the KMP implementation shown in this book is slightly faster than the built-in `C strstr`, `C++ string.find` and `Java String.indexOf`. If an even faster string matching algorithm is needed during contest time for one longer string and much more queries, we suggest using Suffix Array discussed in Section 6.8. There are several other string matching algorithms that are not discussed yet like **Boyer-Moore's**, **Rabin-Karp's**, **Aho-Corasick's**, **Finite State Automata**, etc. Interested readers are welcome to explore them.

We have expanded the discussion of *non classical* DP problems involving string in Section 6.5. We feel that the classical ones will be rarely asked in modern programming contests.

The practical implementation of Suffix Array (Section 6.6) is inspired mainly from the article "Suffix arrays - a programming contest approach" by [68]. We have integrated and synchronized many examples given there with our way of writing Suffix Array implementation. In the third edition of this book, we have (re-)introduced the concept of terminating character in Suffix Tree and Suffix Array as it simplifies the discussion. It is a good idea to solve *all* the programming exercises listed in Section 6.6 although they are not that many *yet*. This is an important data structure that will be more popular in the near future.

Compared to the first two editions of this book, this chapter has grown even more—similar case as with Chapter 5. However, there are several other string processing problems that we have not touched yet: **Hashing Techniques** for solving some string processing problems, the **Shortest Common Superstring** problem, **Burrows-Wheeler transformation** algorithm, **Suffix Automaton**, **Radix Tree**, etc.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	10	24 (+140%)	35 (+46%)
Written Exercises	4	24 (+500%)	17+16* = 33 (+38%)
Programming Exercises	54	129 (+138%)	164 (+27%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
6.3	**Ad Hoc Strings Problems**	126	77%	8%
6.4	String Matching	13	8%	1%
6.5	**String Processing with DP**	17	10%	1%
6.6	Suffix Trie/Tree/Array	8	5%	≈ 1%

L-R: Dr Bill Poucher, Steven
COACH
STEVEN HALIM
ICPC 2012 WORLD FINALS