Lecture 3
Strings

Euiseong Seo  
(euiseong@skku.edu)
Strings

- One of the fundamental data structures
  - Internet search engines
  - Computational biology
  - Processing multimedia data
  - Big data analysis

- One of the plain data structures to manipulate
  - Usually linear
  - Easy to be parallelized

- In C, a string is an array of characters + null
Character Code

- Character codes are mappings between numbers and symbols, which make up a particular alphabet

- ASCII code
  - American Standard Code for Information Exchange
  - 7 bit code, thus $2^7$ characters can be represented
  - The highest order bit is always left as zero
  - De-facto-standard for alpha-numeric character representation
## ASCII Table

<table>
<thead>
<tr>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
<th>DEC</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>1</td>
<td>SOH</td>
<td>2</td>
<td>STX</td>
<td>3</td>
<td>ETX</td>
<td>4</td>
<td>EOT</td>
<td>5</td>
<td>ENQ</td>
<td>6</td>
<td>ACK</td>
<td>7</td>
<td>BEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>9</td>
<td>HT</td>
<td>10</td>
<td>NL</td>
<td>11</td>
<td>VT</td>
<td>12</td>
<td>NP</td>
<td>13</td>
<td>CR</td>
<td>14</td>
<td>SO</td>
<td>15</td>
<td>SI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
<td>17</td>
<td>DC1</td>
<td>18</td>
<td>DC2</td>
<td>19</td>
<td>DC3</td>
<td>20</td>
<td>DC4</td>
<td>21</td>
<td>NAK</td>
<td>22</td>
<td>SYN</td>
<td>23</td>
<td>ETB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
<td>25</td>
<td>EM</td>
<td>26</td>
<td>SUB</td>
<td>27</td>
<td>ESC</td>
<td>28</td>
<td>FS</td>
<td>29</td>
<td>GS</td>
<td>30</td>
<td>RS</td>
<td>31</td>
<td>US</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>SP</td>
<td>33</td>
<td>!</td>
<td>34</td>
<td>&quot;</td>
<td>35</td>
<td>#</td>
<td>36</td>
<td>$</td>
<td>37</td>
<td>%</td>
<td>38</td>
<td>&amp;</td>
<td>39</td>
<td>'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>41</td>
<td>)</td>
<td>42</td>
<td>+</td>
<td>43</td>
<td>,</td>
<td>44</td>
<td>.</td>
<td>45</td>
<td>–</td>
<td>46</td>
<td>/</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>49</td>
<td>1</td>
<td>50</td>
<td>2</td>
<td>51</td>
<td>3</td>
<td>52</td>
<td>4</td>
<td>53</td>
<td>5</td>
<td>54</td>
<td>6</td>
<td>55</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>57</td>
<td>9</td>
<td>58</td>
<td>:</td>
<td>59</td>
<td>;</td>
<td>60</td>
<td>&lt;</td>
<td>61</td>
<td>=</td>
<td>62</td>
<td>&gt;</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>65</td>
<td>A</td>
<td>66</td>
<td>B</td>
<td>67</td>
<td>C</td>
<td>68</td>
<td>D</td>
<td>69</td>
<td>E</td>
<td>70</td>
<td>F</td>
<td>71</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>H</td>
<td>73</td>
<td>I</td>
<td>74</td>
<td>J</td>
<td>75</td>
<td>K</td>
<td>76</td>
<td>L</td>
<td>77</td>
<td>M</td>
<td>78</td>
<td>N</td>
<td>79</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>P</td>
<td>81</td>
<td>Q</td>
<td>82</td>
<td>R</td>
<td>83</td>
<td>S</td>
<td>84</td>
<td>T</td>
<td>85</td>
<td>U</td>
<td>86</td>
<td>V</td>
<td>87</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>X</td>
<td>89</td>
<td>Y</td>
<td>90</td>
<td>Z</td>
<td>91</td>
<td>[</td>
<td>92</td>
<td>]</td>
<td>93</td>
<td>\</td>
<td>94</td>
<td>~</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>97</td>
<td>a</td>
<td>98</td>
<td>b</td>
<td>99</td>
<td>c</td>
<td>100</td>
<td>d</td>
<td>101</td>
<td>e</td>
<td>102</td>
<td>f</td>
<td>103</td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>h</td>
<td>105</td>
<td>i</td>
<td>106</td>
<td>j</td>
<td>107</td>
<td>k</td>
<td>108</td>
<td>l</td>
<td>109</td>
<td>m</td>
<td>110</td>
<td>n</td>
<td>111</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>p</td>
<td>113</td>
<td>q</td>
<td>114</td>
<td>r</td>
<td>115</td>
<td>s</td>
<td>116</td>
<td>t</td>
<td>117</td>
<td>u</td>
<td>118</td>
<td>v</td>
<td>119</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td>121</td>
<td>y</td>
<td>122</td>
<td>z</td>
<td>123</td>
<td>{</td>
<td>124</td>
<td></td>
<td>125</td>
<td>}</td>
<td>126</td>
<td>~</td>
<td>127</td>
<td>DEL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Properties of ASCII

- Several properties of the design make programming easier
  - First three bits are zeros or all seven bits are ones = non-printable characters
  - Both the upper- and lowercase letters and the numerical digits appear sequentially
  - We can convert a character (say, “I”) to its rank in the collating sequence
  - We can convert (say “C”) from upper- to lowercase by adding the difference of the upper and lowercase starting character (“C”-“A”+“a”)}
Properties of ASCII

• A character x is uppercase if and only if it lies between “A” and “Z”
• The character code tells us what will happen when naively sorting text files
• Non-printable character codes for new-line (10) and carriage return (13) are designed to delimit the end of text lines
  – LF+CR vs. LF
Unicode

- Two or three bytes for a symbol
- For any symbol in every language on earth
- ASCII remains alive embedded in Unicode
  - Programs written in old languages can run correctly
  - The upper bytes are all zeros for ASCII/Latin-1
- Java uses 16-bit Unicode representation
  - A character is always 16-bit long
String Representations

- Strings are sequences of characters
  - Order clearly matters

- Several different representations
  - Null-terminated arrays
    - C or C++ default
    - Failing to terminate a string with null typically extends it by a bunch of unprintable characters
  - Array + length
    - Java
    - No need for null character
  - Linked list of characters
    - High space overhead
    - Easy substring replacement
String Representation Choice

- A big impact on which operations are easily or efficiently supported

- Points to consider
  - Which uses the least amount of space? On what sized strings?
  - Which allow constant-time access to the i-th character?
  - Which allow efficient checks that the i-th character is in fact within the string?
  - Which allow efficient deletion or insertion of new characters at the i-th position?
  - Which representation is used when users are limited to strings of length at most 255?
String Libraries

- Many languages provide string manipulation packages
- Don’t reinvent the wheel
String Libraries for C

#include <ctype.h> /* include the character library */

int isalpha(int c); /* true if c is either upper or lower case */
int isupper(int c); /* true if c is upper case */
int islower(int c); /* true if c is lower case */
int isdigit(int c); /* true if c is a numerical digit (0-9) */
int ispunct(int c); /* true if c is a punctuation symbol */
int isxdigit(int c); /* true if c is a hexadecimal digit (0-9,A-F) */
int isprint(int c); /* true if c is any printable character */

int toupper(int c); /* convert c to upper case -- no error checking */
int tolower(int c); /* convert c to lower case -- no error checking */
String Libraries for C

#include <string.h> /* include the string library */

char *strcat(char *dst, const char *src); /* concatenation */
int strcmp(const char *s1, const char *s2); /* is s1 == s2? */
char *strcpy(char *dst, const char *src); /* copy src to dist */
size_t strlen(const char *s); /* length of string */
char *strstr(const char *s1, const char *s2); /* search for s2 in s1 */
char *strtok(char *s1, const char *s2); /* iterate words in s1 */
String Class for C

```c
string::size()    /* string length */
string::empty()   /* is it empty */
string::c_str()   /* return a pointer to a C style string */

string::operator [](size_type i)    /* access the ith character */

string::append(s)   /* append to string */
string::erase(n,m)   /* delete a run of characters */
string::insert(size_type n, const string&s) /* insert string s at n */

string::find(s)
string::rfind(s)    /* search left or right for the given string */

string::first()
string::last()      /* get characters, also there are iterators */
```

C Library String Functions

C contains both character and string libraries. The C language character library contains several simple tests and manipulations on character codes. As with all C predicates, true is defined as any non-zero quantity, and false as zero.

```c
#include <ctype.h> /* include the character library */

int isalpha(int c); /* true if c is either upper or lower case */
int isupper(int c); /* true if c is upper case */
int islower(int c); /* true if c is lower case */
int isdigit(int c); /* true if c is a numerical digit (0-9) */
int ispunct(int c); /* true if c is a punctuation symbol */
int isxdigit(int c); /* true if c is a hexadecimal digit (0-9,A-F) */
int isprint(int c); /* true if c is any printable character */
int toupper(int c); /* convert c to upper case -- no error checking */
int tolower(int c); /* convert c to lower case -- no error checking */
```

Check the definition of each carefully before assuming it does exactly what you want. The following functions appear in the C language string library.

```c
#include <string.h> /* include the string library */

char *strcat(char *dst, const char *src); /* concatenation */
int strcmp(const char *s1, const char *s2); /* is s1 == s2? */
char *strcpy(char *dst, const char *src); /* copy src to dist */
size_t strlen(const char *s); /* length of string */
char *strstr(const char *s1, const char *s2); /* search for s2 in s1 */
char *strtok(char *s1, const char *s2); /* iterate words in s1 */
```

C++ String Library Functions

In addition to supporting C-style strings, C++ has a string class which contains methods for these operations and more:

```c
string::size()    /* string length */
string::empty()   /* is it empty */
string::c_str()   /* return a pointer to a C style string */

string::operator [](size_type i)    /* access the ith character */

string::append(s)   /* append to string */
string::erase(n,m)   /* delete a run of characters */
string::insert(size_type n, const string&s) /* insert string s at n */

string::find(s)
string::rfind(s)    /* search left or right for the given string */

string::first()
string::last()      /* get characters, also there are iterators */
```

Java String Objects

Java strings are first-class objects deriving either from the String class or the StringBuffer class. The String class is for static strings which do not change, while StringBuffer is designed for dynamic strings. Recall that Java was designed to support Unicode, so its characters are 16-bit entities.

The java.text package contains more advanced operations on strings, including routines to parse dates and other structured text.
String Class for Java

- Strings are first-class objects deriving either from the `String` class or the `StringBuffer` class
- String for static string
- StringBuffer for dynamic string
Problem Example: Corporate Renaming

- Company names frequently change
  - For bankruptcy
  - To change notorious image
  - To increase stock price
  - For M&A

- Examples
  - Anderson Consulting to Accenture
  - Enron to Dynegy
  - DEC to Compaq
  - TWA to American

- Our goal is to replace the original names in a text to the new ones
Input/Output Example

3.3 Program Design Example: Corporate Renamings

Corporate name changes are occurring with ever greater frequency, as companies merge, buy each other out, try to hide from bad publicity, or even raise their stock price – remember when adding a .com to a company's name was the secret to success!

These changes make it difficult to figure out the current name of a company when reading old documents. Your company, Digiscam (formerly Algorist Technologies), has put you to work on a program which maintains a database of corporate name changes and does the appropriate substitutions to bring old documents up to date.

Your program should take as input a file with a given number of corporate name changes, followed by a given number of lines of text for you to correct. Only exact matches of the string should be replaced. There will be at most 100 corporate changes, and each line of text is at most 1,000 characters long. A sample input is —

4
"Anderson Consulting" to "Accenture"
"Enron" to "Dynegy"
"DEC" to "Compaq"
"TWA" to "American"

5
Anderson Accounting begat Anderson Consulting, which offered advice to Enron before it DECLARED bankruptcy, which made Anderson Consulting quite happy it changed its name in the first place!

Which should be transformed to —

Anderson Accounting begat Accenture, which offered advice to Dynegy before it CompaqLARED bankruptcy, which made Anderson Consulting quite happy it changed its name in the first place!
Your Plan

- **Read the M&A list into a DB**
  - define DB structure
  - how to handle the double quote ("\") sign?

- **main loop**
  - compare each line of a doc with each DB entry
  - if there is a match, replace it with a new name

- **Now, define**
  - functions
  - global variables
What kind of string operations do we need to do to solve this problem? We must be able to read strings and store them, search strings for patterns, modify them, and finally print them.

Observe that the input file has been segmented into two parts. The first section, the dictionary of name changes, must be completely read and stored before starting to convert the text. To declare the relevant data structures:

```c
#include <string.h>
#define MAXLEN 1001 /* longest possible string */
#define MAXCHANGES 101 /* maximum number of name changes */

typedef char string[MAXLEN];

string mergers[MAXCHANGES][2]; /* store before/after corporate names */
int nmergers; /* number of different name changes */
```
3.4. Searching for Patterns

The simplest algorithm to search for the presence of pattern string \( p \) in text \( t \) overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character:

```c
/* Return the position of the first occurrence of the pattern p in the text t, and -1 if it does not occur. */
int findmatch(char *p, char *t)
{
    int i, j; /* counters */
    int plen, tlen; /* string lengths */
    plen = strlen(p);
    tlen = strlen(t);
    for (i=0; i<=(tlen-plen); i=i+1) {
        j=0;
        while ((j<plen) && (t[i+j]==p[j])) j = j+1;
        if (j == plen) return(i);
    }
    return(-1);
}
```

Note that this routine only searches for exact pattern matches. If a letter is capitalized in the pattern but not in the text there is no match. More seriously, if a company name is split between lines (see the example input), no match will be detected. Such searches can be performed by changing the text/pattern comparison from \( t[i+j]==p[j] \) to something more interesting. The same technique can be used to allow for wild card characters, which match anything. A more general notion of approximate string matching is discussed in Section 11.

This naive algorithm can take as much as \( O(|p||t|) \) time in the worst case. Can you construct an arbitrary-length example pattern and text where it actually takes this much time without ever matching the pattern? Usually the naive search will be much more efficient, since we advance in the text soon as we get a single mismatch. More complicated, linear-time search algorithms do exist: see [Gus97] for a complete
3.4. Searching for Patterns

The simplest algorithm to search for the presence of pattern string $p$ in text $t$ overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character:

```c
// Return the position of the first occurrence of the pattern p
// in the text t, and -1 if it does not occur.
*

int findmatch(char *p, char *t)
{
    int i,j; /* counters */
    int plen, tlen; /* string lengths */

    plen = strlen(p);
    tlen = strlen(t);

    for (i=0; i<=(tlen-plen); i=i+1) {
        j=0;
        while ((j<plen) && (t[i+j]==p[j]))
            j = j+1;
        if (j == plen) return(i);
    }
    return(-1);
}
```

Note that this routine only searches for exact pattern matches. If a letter is capitalized in the pattern but not in the text there is no match. More seriously, if a company name is split between lines (see the example input), no match will be detected. Such searches can be performed by changing the text/pattern comparison from $t[i+j]==p[j]$ to something more interesting. The same technique can be used to allow for wild card characters, which match anything. A more general notion of approximate string matching is discussed in Section 11.

This naive algorithm can take as much as $O(|p|+|t|)$ time in the worst case. Can you construct an arbitrary-length example pattern and text where it actually takes this much time without ever matching the pattern? Usually the naive search will be much more efficient, since we advance in the text soon as we get a single mismatch.

More complicated, linear-time search algorithms do exist: see [Gus97] for a complete String Matching
### Name Replacement

```c
/* Replace the substring of length xlen starting at position pos in string s with the contents of string y. */

replace_x_with_y(char *s, int pos, int xlen, char *y)
{
    int i; /* counter */
    int slen, ylen; /* lengths of relevant strings */

    slen = strlen(s);
    ylen = strlen(y);

    if (xlen >= ylen)
        for (i=(pos+xlen); i<=slen; i++) s[i+(ylen-xlen)] = s[i];
    else
        for (i=slen; i>=(pos+xlen); i--) s[i+(ylen-xlen)] = s[i];

    for (i=0; i<ylen; i++) s[pos+i] = y[i];
}
```
Main Routine

```c
main()
{
  string s;   /* input string */
  char c;     /* input character */
  int nlines; /* number of lines in text */
  int i,j;    /* counters */
  int pos;    /* position of pattern in string */

  read_changes();
  scanf("%d\n",&nlines);
  for (i=1; i<=nlines; i=i+1) { /* read text line */
    j=0;
    while ((c=getchar()) != '\n') {
      s[j] = c;
      j = j+1;
    }
    s[j] = '\0';

    for (j=0; j<nmergers; j=j+1)
      while ((pos=findmatch(mergers[j][0],s)) != -1) {
        replace_x_with_y(s, pos,
                         strlen(mergers[j][0]), mergers[j][1]);
      }

    printf("%s\n",s);
  }
}
```

Example Review

- Error and exception handling
- Efficiency
  - Space complexity
  - Time complexity
A popular but insecure method of encrypting text is to permute the letters of the alphabet. That is, in the text, each letter of the alphabet is consistently replaced by some other letter. To ensure that the encryption is reversible, no two letters are replaced by the same letter.

A powerful method of cryptanalysis is the known plain text attack. In a known plain text attack, the cryptanalyst manages to have a known phrase or sentence encrypted by the enemy, and by observing the encrypted text then deduces the method of encoding.

Your task is to decrypt several encrypted lines of text, assuming that each line uses the same set of replacements, and that one of the lines of input is the encrypted form of the plain text the quick brown fox jumps over the lazy dog.

---

Sample Input

1
vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq
xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj
frtjrpgguvj otvxmdxd prm iev prmvx xnmq

Sample Output

now is the time for all good men to come to the aid of the party
the quick brown fox jumps over the lazy dog
programming contests are fun aren’t they
Crypt Kicker II

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive cases.

Each case consists of several lines of input, encrypted as described above. The encrypted lines contain only lowercase letters and spaces and do not exceed 80 characters in length. There are at most 100 input lines.

Output

For each test case, decrypt each line and print it to standard output. If there is more than one possible decryption, any one will do. If decryption is impossible, output No solution.

The output of each two consecutive cases must be separated by a blank line.
A popular but insecure method of encrypting text is to permute the letters of the alphabet. That is, in the text, each letter of the alphabet is consistently replaced by some other letter. To ensure that the encryption is reversible, no two letters are replaced by the same letter.

A powerful method of cryptanalysis is the known plain text attack. In a known plain text attack, the cryptanalyst manages to have a known phrase or sentence encrypted by the enemy, and by observing the encrypted text then deduces the method of encoding. Your task is to decrypt several encrypted lines of text, assuming that each line uses the same set of replacements, and that one of the lines of input is the encrypted form of the plain text

the quick brown fox jumps over the lazy dog.

**Input**

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive cases.

Each case consists of several lines of input, encrypted as described above. The encrypted lines contain only lowercase letters and spaces and do not exceed 80 characters in length. There are at most 100 input lines.

**Output**

For each test case, decrypt each line and print it to standard output. If there is more than one possible decryption, any one will do. If decryption is impossible, output No solution.

The output of each two consecutive cases must be separated by a blank line.

**Sample Input**

1

vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj frtjrpgguvj otvxdxd prm iev prmvx xnmq

**Sample Output**

now is the time for all good men to come to the aid of the party
the quick brown fox jumps over the lazy dog
programming contests are fun aren't they
Doublets

A doublet is a pair of words that differ in exactly one letter; for example, “booster” and “rooster” or “rooster” and “roaster” or “roaster” and “roasted”.

You are given a dictionary of up to 25,143 lowercase words, not exceeding 16 letters each. You are then given a number of pairs of words. For each pair of words, find the shortest sequence of words that begins with the first word and ends with the second, such that each pair of adjacent words is a doublet. For example, if you were given the input pair “booster” and “roasted”, a possible solution would be (“booster,” “rooster,” “roaster,” “roasted”), provided that these words are all in the dictionary.

Input

The input file contains the dictionary followed by a number of word pairs. The dictionary consists of a number of words, one per line, and is terminated by an empty line. The pairs of input words follow; each pair of words occurs on a line separated by a space.

Output

For each input pair, print a set of lines starting with the first word and ending with the last. Each pair of adjacent lines must be a doublet.

If there are several minimal solutions, any one will do. If there is no solution, print a line: “No solution.” Leave a blank line between cases.
# Doublelets

## Sample Input

booster  
rooster  
roaster  
coasted  
roasted  
coastal  
postal

booster roasted  
coastal postal

## Sample Output

booster  
rooster  
roaster  
roasted  
No solution.