Algorithms and Lab (CSE130)

String Algorithms

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

Contents

Strings

Strings

- A string is a data-type in programming that holds a sequence of characters. More fancily said, "a string is often implemented as an array of bytes (or words) that stores a sequence of elements (typically characters) using some sort of character encoding".
- Depending on the programming language, strings can either be mutable or immutable.
- String manipulations are used for word processing applications such as creating, inserting, updating, and modifying textual data.
- Besides this, we need to search for a particular pattern within a text, delete it, or replace it with another pattern. So, there is a lot that we as users do to manipulate the textual data.
- Common string operations include finding lengths, copying, searching, replacing and counting the occurrences of specific characters and worlds.
- Handling strings and patterns is important and necessary in programming. $https : //www.w3schools.com/python/python_regex.asp$

Strings (cont...)

- String methods in Python

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case

Strings (cont...)

Method	**Description**
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Strings (cont...)

- Problems related to Strings

Longest Common Substring Problem

Find the longest substring of a string containing $k-$distinct characters

Shortest Superstring Problem

Determine whether a string matches with a given pattern

Count the number of times a pattern appears in a given string as a subsequence

Find the minimum number of deletions required to convert a string into a palindrome

Find the longest substring of a string containing distinct characters

Check if strings can be derived from each other by circularly rotating them

Determine whether two strings are anagram or not

Isomorphic Strings

Longest Common Prefix in a given set of strings

Find all N-digit binary strings without any consecutive 1's

Find the maximum occurring word in a given set of strings

Find first $k - maximum$ occurring words in a given set of strings

Print all pairs of anagrams in a set of strings

Find all substrings containing exactly $k - distinct$ characters

Construct the longest palindrome by shuffling or deleting characters from a string

Determine whether a string is a palindrome or not

Check if a string is a rotated palindrome or not

Longest Palindromic Substring Problem

Find all possible palindromic substrings of a string

Find all substrings of a string that are a permutation of another string

Iterative approach to finding permutations of a string

Find all lexicographic permutations of a string

Lexicographic rank of a string

Remove all extra spaces from a string

Remove adjacent duplicate characters from a string

Lexicographically Minimal String Rotation

Determine whether a string can be transformed into another string in a single edit

Find minimum operations required to transform a string into another string

Reverse a string using a stack data structure

Reverse a string without using recursion

Exact String Matching

String Matching Problem

- A string is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters;
- String matching: given a string of n characters called the text and a string of m characters $(m \leq n)$ called the pattern, find a substring of the text that matches the pattern. If the indices for Text $t_0 ... tn - 1$ and Pattern $p_0 ... pm - 1$, then

$$\begin{aligned} Text &= t_0, ..., t_i, ...t_{i+j}, ..., t_{i+m-1}, ..., t_{n-1} \\ Pattern &= \qquad p_0, ..., p_j, ...p_{m-1} \\ substring &= t_i = p_0, ..., t_{i+j} = p_j, ..., t_{i+m-1} = p_{m-1} \end{aligned}$$

- String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems
- Applications of String Matching Algorithms:
  - Plagiarism Detection:
  - Bioinformatics and DNA Sequencing:
  - Digital Forensics:
  - Spelling Checker:
  - Spam filters:
  - Search engines or content search in large databases:
  - Intrusion Detection System:

Exact String Matching (cont...)

- Exact String Matching Algorithms
  - Brute Force algorithm
  - Deterministic Finite Automaton algorithm
  - Karp-Rabin algorithm
  - Shift Or algorithm
  - Morris-Pratt algorithm
  - Knuth-Morris-Pratt algorithm
  - Simon algorithm
  - Colussi algorithm
  - Galil-Giancarlo algorithm
  - Apostolico-Crochemore algorithm
  - Not So Naive algorithm
  - Boyer-Moore algorithm
  - Turbo BM algorithm
  - Apostolico-Giancarlo algorithm
  - Reverse Colussi algorithm
  - Horspool algorithm
  - Quick Search algorithm
  - Tuned Boyer-Moore algorithm
  - Zhu-Takaoka algorithm
  - Berry-Ravindran algorithm
  - Smith algorithm
  - Raita algorithm
  - Reverse Factor algorithm
  - Turbo Reverse Factor algorithm
  - Forward Dawg Matching algorithm
  - Backward Nondeterministic Dawg Matching algorithm
  - Backward Oracle Matching algorithm
  - Galil-Seiferas algorithm
  - Two Way algorithm
  - String Matching on Ordered Alphabets algorithm
  - Optimal Mismatch algorithm
  - Maximal Shift algorithm
  - Skip Search algorithm
  - KMP Skip Search algorithm
  - Alpha Skip Search algorithm
- A comprehensive discussion on these algorithms can be found at $https : //www - igm.univ - mlv.fr/\ lecroq/string/index.html$

Exact String Matching (cont...)

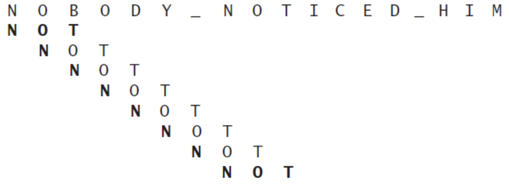
A brute-force algorithm for the string-matching problem

- Input: An array $T[0..n-1]$ of n characters representing a text and an array $P[0..m-1]$ of m characters representing a pattern
- Output: The index of the first character in the text that starts a matching substring, otherwise -1

- Algorithm

```
procedure SM(T[0..n-1], P[0..m-1])
    for $(i = 0; i <= n - m; i++)$ do
        j=0
        while $(j < m$ and $P[j] == T[i + j])$ do
            j=j+1
            if $j == m$ then
                return i
            end if
        end while
    end for
    return -1
end procedure
```

- Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.



```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

$$T(n, m) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1 = (n - m + 1)m = nm - m^2 + 1 \in \Theta(nm)$$

Horspool's algorithm

Horspool's algorithm

- Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern
- Knuth-Morris-Pratt (KMP): algorithm preprocesses pattern left to right to get useful information for later searching
- Boyer -Moore algorithm: preprocesses pattern right to left and store information into two tables
- Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table
- Consider, as an example, searching for the pattern BARBER in some text:

$$\begin{array}{ccccc} s_0 & \ldots & c & \ldots & s_{n-1} \\ & & \mathit{BARBER} & & \end{array}$$

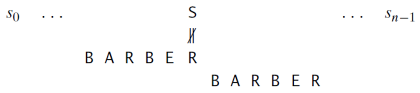
- Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found.
- If a mismatch occurs, we need to shift the pattern to the right.

Horspool's algorithm (cont...)

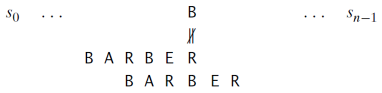
- Horspool's algorithm determines the size of such a shift by looking at the character *c* of the text that is aligned against the last character of the pattern.

- In general, the following four possibilities can occur.

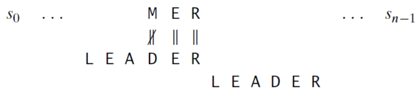
  1. If there are no c's in the pattern: $\rightarrow$ shift the pattern by its entire length



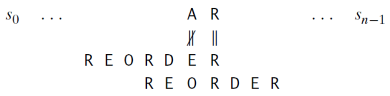
  2. If there are occurrences of character c in the pattern but it is not the last one there: $\rightarrow$ shift should align the rightmost occurrence of c in the pattern with the c in the text



  3. If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters: $\rightarrow$ shift the pattern by its entire length


s_0 . . . M E R . . . s_{n-1}
$\not\parallel$ $\parallel$ $\parallel$
L E A D E R
L E A D E R

  4. if c happens to be the last character in the pattern and there are other c's among its first $m-1$ characters: $\rightarrow$ shift should align the rightmost occurrence of c in the pattern with the c in the text



Horspool's algorithm (cont...)

Algorithm for computing shift table

- A simple algorithm for computing the shift table entries that initializes all the entries to the pattern's length *m* and scan the pattern left to right repeating the following step $m-1$ times:
- Input: Pattern P[0..m $-$ 1] and an alphabet of possible characters
- Output: Table[0..size $-$ 1] indexed by the alphabet's characters and filled with shift sizes computed by formula
- Algorithm

```
1: procedure SHIFTTABLE(P[0..m-1])
2:     for $(i = 0; i <= size - 1; i++)$ do
3:         Table[i]=m
4:     end for
5:     for $(j = 0; i <= m - 2; j++)$ do
6:         table[i]=m-j-1
7:     end for
8:     return Table
9: end procedure
```

- Complexity $T(n) \in O(m)$

Horspool's algorithm (cont...)

Horspool's algorithm

- Step 1 For a given pattern of length m, construct the shift table.
- Step 2 Align the pattern against the beginning of the text.
- Step 3 Repeat the following until either pattern matches or it reaches beyond the last character of the text.
  1. Starting with the last character in the pattern, compare the corresponding characters in the pattern
  2. If a mismatching pair is occurred,, retrieve the entry t (c) from the c's column of the shift table and shift the pattern by t(c) characters to the right along the text.
- Algorithm

```
procedure HorspoolMatching(T[0..n-1], P[0..m-1])
    Table=ShiftTable(P [0..m  1])                    ▷ generate Table of shifts
    i=m-1                                            ▷ position of the pattern's right end
    while $(i \leq n-1)$ do
        k=0                                          ▷ number of matched characters
        while $(k \leq m-1)$ and $(P[m-1-k] = T[i-k])$ do
            k=k + 1
            if (k = m) then return i − m + 1
            else  $i = i + Table[T[i]]$
            end if
        end while
    end while
    return -1
end procedure
```

- Complexity the worst-case efficiency of Horspool's algorithm is in $\Theta(nm)$. But for random texts, it is in $\Theta(n)$,

Horspool's algorithm (cont...)

- EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

Knuth-Morris-Pratt algorithm

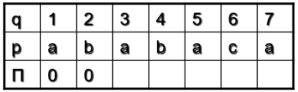
Knuth-Morris-Pratt algorithm

- Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.
- A matching time of O(n) is achieved by avoiding comparisons with elements of ‘T’ that have previously been involved in comparison with some element of the pattern ‘P’ to be matched.
- The prefix function, Π : The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.
- This information can be used to avoid useless shifts of the pattern ‘P’.
- Some times prefix table is also known as LPS Table. Here LPS stands for ”Longest proper Prefix which is also Suffix”.
- Complexity: For computing the prefix function, the for loop from step 4 to step 10 runs ‘m’ times, the running time of compute prefix function is $O(m)$.

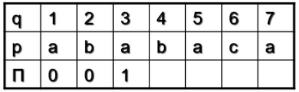
```
1:  procedure LPS(P[1..m])
2:      $m = len(P)$
3:      $\Pi[1] = 0$ , $k = 0$
4:      for (for $q = 2$ to $m$) do
5:          while $k > 0$ and $P[k+1]! = P[q]$ do
6:              $k = \Pi[k]$
7:              if $(P[k+1] = P[q])$ then
8:                  $k = k + 1$
9:              end if
10:             $\Pi[k] = k$
11:         end while
12:     end for
13:     return $\Pi$
14: end procedure
```

Knuth-Morris-Pratt algorithm (cont...)

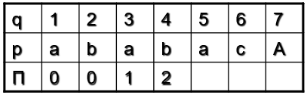
- Compute Π for the pattern P: [a b a b a c a]
- Initially: m = length[p] = 7, Π[1] = 0, k = 0
- Step 1: q = 2, k=0 , Π[2] = 0
- Step 2: q = 3, k = 0, Π[3] = 1
- Step 3: q = 4, k = 1, Π[4] = 2
- Step 4: q = 5, k =2, Π[5] = 3
- Step 5: q = 6, k = 3, Π[6] = 1
- Step 6: q = 7, k = 1 ,Π[7] = 1



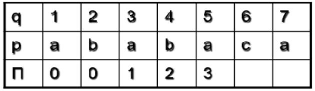
q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					



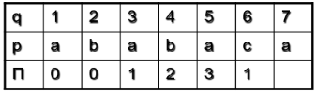
q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				



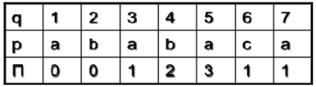
q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			



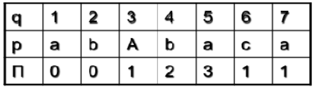
q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		



q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	



q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1



q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	1	1

Knuth-Morris-Pratt algorithm (cont...)

- The KMP Matcher: With string ‘T’, pattern ‘P’ and prefix function Π as inputs, finds the occurrence of ‘p’ in ‘T’ and returns the number of shifts of ‘P’ after which occurrence is found.

```
 1: procedure KMP-Matcher(T[1..n]P[1..m])
 2:     $n = len(T)$
 3:     $m = len(P)$
 4:     $\Pi[1] = LPS(P)$
 5:     $q = 0$                                          ▷ number of characters matched
 6:     for (for $i = 1$ to $n$) do                      ▷ scan T from left to right
 7:         while $q > 0$ and $P[q+1]! = S[i]$ do
 8:             $q = \Pi[q]$                             ▷ next character does not match
 9:             if $(P[q+1] = S[i])$ then
10:                 $q = q + 1$                          ▷ next character matches
11:             end if
12:             if $(q = m)$ then                        ▷ P matched
13:                 print Pattern
14:             end if
15:             $q = \Pi[q]$                             ▷ look for the next match
16:         end while
17:     end for
18: end procedure
```

- Complexity: The for loop beginning in step 5 runs ‘n’ times, i.e., as long as the length of the string ‘T’. Thus running time of matching function is $O(n)$.

Approximate String Matching

Sequence Alignment

- A sequence alignment is a way of placing one sequence above the other in order to identify the correspondence between similar characters or substrings.
- It can be performed on Deoxyribonucleic acid (DNA), Ribonucleic acid (RNA) or protein sequences.
- Sequences from different organisms may be of different sizes. An alignment requires the insertion of spaces in arbitrary locations along the sequence so that both will have the same size. 'Spaces' or 'gaps' are inserted either in the beginning or in the end of the sequences.
- For example consider the two strings:

$$X = \texttt{AGGCTATCACCTGACCTCCAGGCCGATGCCC}$$
$$Y = \texttt{TAGCTATCACGACCGCGGTCGATTTGCCCGAC}$$

We can align these strings as follows:

$$X' = \texttt{-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC--}$$
$$Y' = \texttt{TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC}$$

- Types of alignment: There are two types of alignment based on what we are looking for:
  1. Global alignment: Aligns a query sequence with the target sequence along the entire length of the sequence. Global alignment is a global optimization process that spans the entire length of two query sequences.
  2. Local alignment: Aligns a substring of the query sequence to a substring of the target sequence, i.e. finding local regions of high similarity between two sequences.
- There are also two types of alignment based on the number of sequences being aligned
  1. Pairwise alignment: Involves two sequences, the query and the target with which it is aligned.
  2. Multiple alignment: Involves alignment with more than two sequences.

Edit Distance

Problem Statement

- Aligning Sequences without Insertions and Deletions:
  Hamming Distance: $d_H(v, w) = 8$

```
v : ATATATAT
w : TATATATA
```

- Aligning Sequences with Insertions and Deletions
  $d(v, w) = 2$

```
v : ATATATAT--
w : --TATATATA
```

  $d(v, w)$= MIN number of elementary operations to transform $v \to w$

- Edit Distance Levenshtein (1966) introduced edit distance between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other

- Example: TGCATAT $\to$ ATCCGAT in 5 steps
  TGCATAT $\to$ (delete last T)
  TGCATA $\to$ (delete last A)
  TGCAT $\to$ (insert A at front)
  ATGCAT $\to$ (substitute C for 3rd G)
  ATCCAT $\to$(insert G before last A)
  ATCCGAT $\to$ (Already Done)

subproblems

- Suppose we have an optimal alignment for two sequences S and T in which S_i matches T_j.

- The key insight is that this optimal alignment is composed of an optimal alignment between $(S_1, \ldots, S_{i-1})$ and $(T_1, \ldots, T_{j-1})$ and an optimal alignment between $(S_{i+1}, \ldots, S_n)$ and $(T_{j+1}, \ldots, T_m)$.

Edit Distance (cont...)

Local optimality

- We can compute the optimal solution for a subproblem by making a locally optimal choice based on the results from the smaller sub-problems.
- Thus, we need to establish a recursive function that shows how the solution to a given problem depends on its subproblems.
- We use this recursive definition to fill up the table F in a bottom-up fashion.
- Let $F_{i,j}$ be the score of the optimal alignment of $(S_1, \ldots, S_i)$ and $(T_1, \ldots, T_j)$. The space of subproblems is $\{F_{i,j}\ \forall i \in [1, |S|], \forall j \in [1, |T|]\}$.

$$F(i,j) = \begin{cases} F(0,0) = 0 & i = 0, j = 0 \\ F(i,0) = i & 1 \leq i \leq m \\ F(0,j) = j & 1 \leq j \leq n \\ min \begin{cases} F(i-1,j) + 1, & (insert) \\ F(i,j-1) + 1, & (delete) \\ cost(X_i, Y_j) + F(i-1,j-1) & (copy/substitute) \end{cases} & \end{cases}$$

where if $X_i == Y_j$ then $cost(X_i, Y_j) = 0$ otherwise $cost(X_i, Y_j) = 1$

Longest Common Subsequence

Longest Common Subsequence

- A **subsequence** is any subset of the elements of a sequence that maintains the same relative order. If A is a subsequence of B, this is denoted by $A \subset B$.
- For example, if $A = a_1 a_2 a_3 a_4 a_5$, the sequence $A' = a_2 a_4 a_5$ is a subsequence of A since the elements appear in order, while the sequence $A'' = a_2 a_1 a_5$ is not a subsequence of A since the elements a_2 and a_1 are reversed.
- The Longest Common Subsequence (LCS) of two sequences X and Y is a sequence L such that $L \subset X$, $C \subset Y$, and $|L|$ is maximized.
- Example
  $A = abacdac$; $B = cadcddc$; $C = acdc$
- In this example, C is a LCS of A and B since it is a subsequence of both and there are no subsequences of length 5.
- if |X| = m, |Y| = n, then there are 2m subsequences of x; we must compare each with Y (n comparisons) So the running time of the brute-force algorithm is $O(n2^m)$

Longest Common Subsequence (cont...)

Optimal Substructure

- Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y.
  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
  2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
  3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.
- Recursive Formula Let $L[i, j]$ be the length of the LCS of X_i and Y_j (prefixes).

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ L[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(L[i, j-1], L[i-1, j]) & \text{otherwise.} \end{cases}$$

- Recursive Algorithm

```
procedure LCS-DC(X, Y, m, n)
    if ($m == 0$ and $n == 0$) then
        return 0
    end if
    if ($X[m-1] == Y[n-1]$) then
        return 1 + LCS-DC(X, Y, m-1, n-1)
    else
        return max(LCS-DC(X, Y, m, n-1), LCS-DC(X, Y, m-1, n))
    end if
end procedure
```

- Complexity $T(n) \in O(2^n)$

Longest Common Subsequence (cont...)

- Dynamic Programming based Algorithm

```
procedure LCSDP(X,Y)
    m =len(X), n=len(Y)
    for each $i \in m$   $L[i, 0] = 0$
    for each $j \in n$   $L[0, j] = 0$
    for $(i = 1\ to\ m)$ do
        for $(j = 1\ to\ n)$ do
            if $(x_i = y_j)$ then
                L[i,j] = L[i-1,j-1]+1
                B[i,j] = 'D'
            else
                if $(L[i-1, j] \geq L[i, j-1])$ then
                    L[i,j] = L[i-1,j]
                    B[i,j] = 'U'
                else
                    L[i,j] = L[i,j-1]
                    B[i,j] = 'L'
                end if
            end if
        end for
    end for
    return L,B
end procedure
```

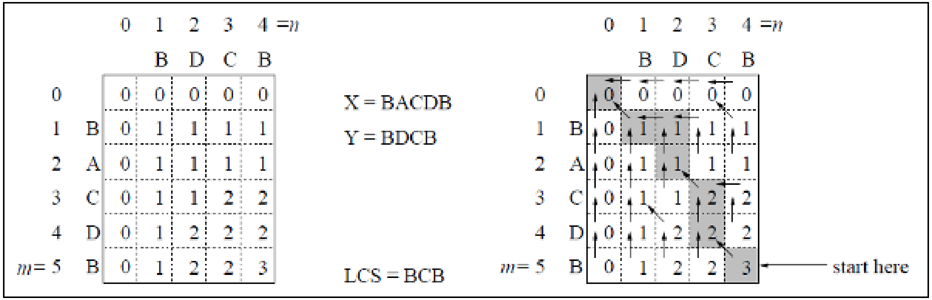
- Complexity: To populate the table, the outer for loop iterates m times and the inner for loop iterates n times. Hence, the complexity of the algorithm is O(m, n), where m and n are the length of two strings.

- Algorithm for printing LCS from table B

```
procedure Print-LCS(B, X, i, j)
    if $(i == 0$ and $j == 0)$ then
        return
    else
        if $(B[i, j] =' D')$ then
            Print-LCS(B, X, i-1, j-1)
        else
            if $B[i, j] =' U'$ then
                Print-LCS(B, X, i-1, j)
            else
                Print-LCS(B, X, i, j-1)
            end if
        end if
    end if
end procedure
```

Longest Common Subsequence (cont...)

- Example: Two strings X = BACDB and Y = BDCB to find the longest common subsequence.
- Following the algorithm LCS-DP, we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).



		0	1	2	3	4 $=n$
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m=5$	B	0	1	2	2	3

X = BACDB

Y = BDCB

LCS = BCB

		0	1	2	3	4 $=n$
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m=5$	B	0	1	2	2	3

- In table B, instead of 'D', 'L' and 'U' we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.

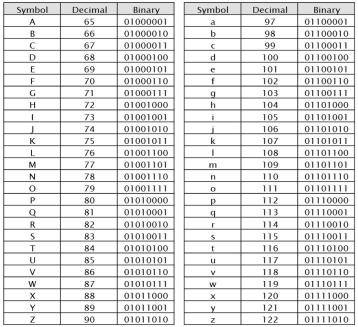
String Compression

String Compression

- The problem of **string compression** is to find an efficient method for encoding a text data. A character encoding tells the computer how to interpret raw zeroes and ones into real characters.
- American Standard Code for Information Interchange (ASCII), Universal Multiple-Octet Coded Character Set (UCS) (UCS-2), UCS Transformation Format 8 (UTF-8), UCS Transformation Format 16 (UTF-16) are examples for encodings schemes

fixed-length binary code

- A common way to represent a file is to use a binary code. In such a code, each character is represented by a unique binary string, called the **codeword**.
- A fixed-length binary code represents each character using the same numbers of bits.
- ASCII codes shown in Figure 5 are fixed-length binary codes
- No of bits to encode the string "Muhammad Tariq Mahmood" $\rightarrow 8 \times 22 = 176$ *bits*



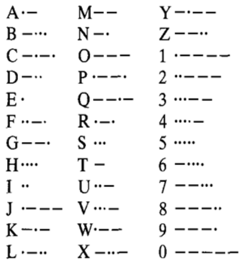
Symbol	Decimal	Binary	Symbol	Decimal	Binary
A	65	01000001	a	97	01100001
B	66	01000010	b	98	01100010
C	67	01000011	c	99	01100011
D	68	01000100	d	100	01100100
E	69	01000101	e	101	01100101
F	70	01000110	f	102	01100110
G	71	01000111	g	103	01100111
H	72	01001000	h	104	01101000
I	73	01001001	i	105	01101001
J	74	01001010	j	106	01101010
K	75	01001011	k	107	01101011
L	76	01001100	l	108	01101100
M	77	01001101	m	109	01101101
N	78	01001110	n	110	01101110
O	79	01001111	o	111	01101111
P	80	01010000	p	112	01110000
Q	81	01010001	q	113	01110001
R	82	01010010	r	114	01110010
S	83	01010011	s	115	01110011
T	84	01010100	t	116	01110100
U	85	01010101	u	117	01110101
V	86	01010110	v	118	01110110
W	87	01010111	w	119	01110111
X	88	01011000	x	120	01111000
Y	89	01011001	y	121	01111001
Z	90	01011010	z	122	01111010

String Compression (cont...)

variable-length code

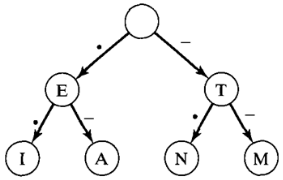
- A variable-length binary code represents different characters using different numbers of bits.
- One such example is Morse code.
- In **Morse code**, characters are represented as sequences of dots and dashes, as shown in the following Figure

Variable length codes



A	·–	M	––	Y	–·––
B	–···	N	–·	Z	––··
C	–·–·	O	–––	1	·––––
D	–··	P	·––·	2	··–––
E	·	Q	––·–	3	···––
F	··–·	R	·–·	4	····–
G	––·	S	···	5	·····
H	····	T	–	6	–····
I	··	U	··–	7	––···
J	·–––	V	···–	8	–––··
K	–·–	W	·––	9	––––·
L	·–··	X	–··–	0	–––––

Tree representation



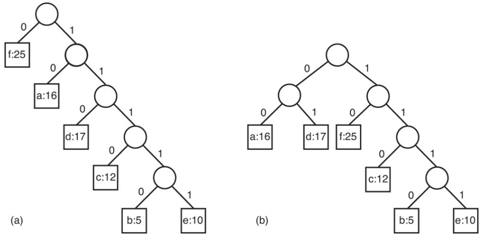
String Compression (cont...)

- Example:
  - Suppose our character set is a,b,c,d,e,f and each character appears in the file the number of times indicated in Table
  - Number of bits for a binary tree can be computed as

    $$Bits\,(T) = \sum_{i=1}^{n} frequency\,(v_i) \times depth\,(v_i)$$

    where; $\{v_1, v_2, \ldots v_n\}$ is the set of characters in the file, $frequency(v_i)$ is the number of times v_i occurs in the file, $depth(v_i)$ is the depth of v_i in tree T.
  - Number of bits for each coding scheme

    $$Bits\,(C_1) = 16\,(3) + 5\,(3) + 12\,(3) + 17\,(3) + 10\,(3) + 25\,(3) = 255$$
    $$Bits\,(C_2) = 16\,(2) + 5\,(5) + 12\,(4) + 17\,(3) + 10\,(5) + 25\,(1) = 231$$
    $$Bits\,(C_3) = 16\,(2) + 5\,(4) + 12\,(3) + 17\,(2) + 10\,(4) + 25\,(2) = 212$$



Character	Frequency	C1(Fixed-Length)	C2	C3(Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Huffman Coding Algorithm

Huffman Coding Algorithm

- Huffman code is a data compression algorithm which uses the greedy technique for its implementation.
- If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's symbols with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1.
- for a given alphabet with known frequencies of the symbol occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency symbols and longer ones to low-frequency symbols?
- Recall, the Optimal Binary Code problem is to find a binary character code for the characters in the file, which represents the file in the least number of bits.
- Huffman developed a greedy algorithm that produces an optimal binary character code by constructing a binary tree corresponding to an optimal code.
- A code produced by this algorithm is called a **Huffman code**
- Huffman's encoding is one of the most important file-compression methods.

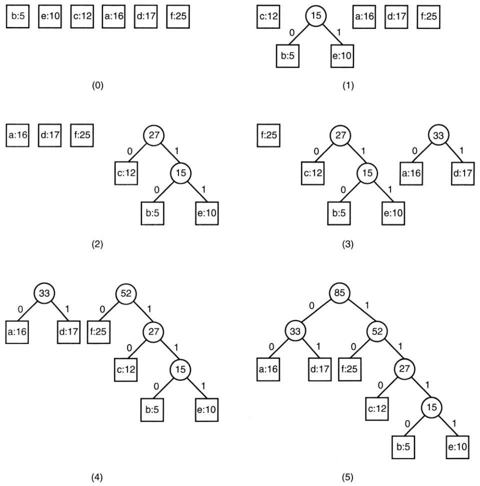
Huffman Coding Algorithm (cont...)

- To implement Huffman Algorithm we need **Node** and **Priority Queue(PQ)** data structures

- Algorithm (Pseudo-code)

```
procedure HUFFMAN(datafile)
    for ($i = 1; i <= n - 1; i++$) do
        $p = remove(PQ)$
        $q = remove(PQ)$
        $r = new\ Node()$
        $r.left = p$
        $r.right = q$
        $r.frequency = p.frequency + q.frequency$
        $insert\ (PQ,\ r)$
    end for
    $r = remove(PQ)$
    return $r$
end procedure
```

- Running Example



b:5 e:10 c:12 a:16 d:17 f:25
(0)
c:12 15 a:16 d:17 f:25 b:5 e:10 0 1
(1)
a:16 d:17 f:25 27 c:12 15 b:5 e:10 0 1
(2)
f:25 27 c:12 15 b:5 e:10 33 a:16 d:17 0 1
(3)
33 a:16 d:17 52 f:25 27 c:12 15 b:5 e:10 0 1
(4)
85 33 52 a:16 d:17 f:25 27 c:12 15 b:5 e:10 0 1
(5)

Summary