CEN360
ALGORITHMS DESIGN AND ANALYSIS
Fall – 2018
Lab 10: Transform and Conquer Algorithms

1. Horner's Rule For Polynomial Evaluation: [20pts]

$p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0$ at a point $x = x_0$

As you have already seen in LAB-5, there are two different brute force algorithms for evaluating a polynomial at a given point.

Brute-force algorithm version 1	Brute-force algorithm version 2 (Evaluating from right to left)
p* = 0.0** **for *i* = *n* downto 0 do** ***power* = 1** **for *j* = 1 to *i* do //compute x^i** ***power* = *power* * *x ***p* = *p* + *a*[*i*] * *power*** ***return p***	***p* = 0.0** ***power* = 1** **for *i* = *1* to n do** ***power* = *power* * *x*** ***p* = *p* + *a*[*i*] * *power*** ***return p***

In this part of the LAB, you are going to implement the Horner's Rule for polynomial evaluation. In the following, there is an example:

$p(x) = 2x^4 - x^3 + 3x^2 + x - 5 =$
$= x(2x^3 - x^2 + 3x + 1) - 5 =$
$= x(x(2x^2 - x + 3) + 1) - 5 =$
$= x(x(x(2x - 1) + 3) + 1) - 5$

Substitution into the last formula leads to a faster algorithm! The pseudo code for Horner's Rule is given as follows:

Horner's Rule for polynomial evaluation	Test Case of example given above
***Algorithm** Horner (P[0..n], x)* *//Evaluates a polynomial at a given point by* *//Horner's rule. **Input:** An array P[0..n] of* *//coefficients of a polynomial of degree n (stored* *//from lowest to the highest) and a number x* *//**Output:** The value of the polynomial at x* `p = P[n]` `for i = n-1 downto 0 do` `  p = x*p + P[i]` `return p`	$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ $p=2*3^4 - 3^3 + 3*3^2 + 3-5$ $=162 - 27 + 27 -2 =$ **160** P = {-5, 1, 3, -1, 2} p=2 x= 3 p=3*2-1=5 p=3*5+3=18 p=3*18+1=55 p=3*55-5=**160**

2. Computing Integer Power a^n: [30pts]

In previous LABs (LAB4 and LAB8), you have implemented Brute force algorithm and Divide and Conquer Algorithms for calculating Integer Power a^n.

Brute-force algorithm

```
power = 1
for i = 1 to n do //compute x^n
    power = power * x
```

Divide and Conquer algorithm

```
Algorithm power(a,n)
  if (n = 1)
    return a
  partial ← power(a,floor(n/2))
  if n mod 2 = 0
    return partial × partial
  else
    return partial × partial × a
```

$$a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & \text{if } n \text{ is even} \\ a^{\frac{n}{2}} \times a^{\frac{n}{2}} \times a & \text{if } n \text{ is odd} \end{cases}$$

Complexity: $T(n) = T(n/2) + O(1) \Rightarrow T(n)$ is $O(\log n)$

In this part of the LAB, you are going to implement the Left-to-right binary exponentiation and right-to-left binary exponentiation algorithms for computing integer power a^n. In the following, examples are given for both algorithms:

Left-to-right binary exponentiation

Initialize product accumulator by 1.

Scan n's binary expansion from left to right and do the following:

If the current binary digit is 0, square the accumulator (S); if the binary digit is 1, square the accumulator and multiply it by a (SM).

Example: Compute a^{13}. Here, $n = 13 = 1101_2$

binary rep. of 13:	1	1	0	1
	SM	SM	S	SM
accumulator: 1	$1^2*a=a$	$a^2*a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2*a= a^{13}$ (computed left-to-right)

Efficiency: $b \le M(n) \le 2b$ where $b = \lfloor \log_2 n \rfloor + 1$

Right-to-left binary exponentiation

Scan n's binary expansion from right to left and compute a^n as the product of terms a^{2^j} corresponding to 1's in this expansion.

Example Compute a^{13} by the right-to-left binary exponentiation. Here, $n = 13 = 1101_2$.

1	1	0	1		
a^8	a^4	a^2	a	:	a^{2^j} terms
a^8 *	a^4 *		a	:	product

(computed right-to-left)

Efficiency: same as that of left-to-right binary exponentiation

3. String searching by preprocessing: [50pts]

In LAB4 (Q5), you have already implemented Brute force algorithm for string searching and matching.

String searching by brute force
pattern*:** a string of *m* characters to search for ***text*:** a (long) string of *n* characters to search in ***Brute force algorithm **Step 1** Align pattern at beginning of text **Step 2** Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected **Step 3** While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

In this part of the LAB, you are going to implement Horspool's Algorithm for String searching by preprocessing.

Firstly, create a text file. Put some sentences and paragraphs inside the file. In the program create a word as a string and check whether your string is in the given file or not. If it is in the file, then print indexes of each occurrence in the file.

Horspool's Algorithm:

- If a mismatch occurs, we need to shift the pattern to the right.
- Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text
- 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

Shift table

- Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

- Shift table is indexed by text and pattern alphabet
  Eg, for `BAOBAB`:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Shift Table Generation:

Algorithm ShiftTable(P[0..m-1])

```
//fill the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0...m-1] and and aalphabet of possible characters
//Output: Table [0..size-1] indexed by the alphabet's characters and filled
//with shift sizes computed by formula

for i=0 to size-1 do
      Table[i] = m
for j=0 to m-2 do
      Table[P[j]] = m-1-j
return Table
```

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])      //generate Table of shifts
i ← m − 1                    //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                    //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```