COMP 3270 Assignment 2 9 problems 100 points 10% Credit
Due before 11:59 PM Monday September 24

Instructions:

1. This is an individual assignment. You should do your own work. Any evidence of copying will result in a zero grade and additional penalties/actions.
2. Enter your answers in this Word file. Submissions must be uploaded **as a single file** (Word or PDF preferred, but other formats acceptable as long as your work is LEGIBLE) to Canvas before the due date and time. Don't turn in photos of illegible sheets. **If an answer is unreadable, it will earn zero points.** Cleanly handwritten submissions (print out this assignment and write answers in the space provided, with additional sheets used if needed) scanned in as PDF and uploaded to Canvas are acceptable.
3. **Submissions by email or late submissions (even by minutes) will receive a zero grade.** No makeup will be offered unless prior permission to skip the assignment has been granted, or there is a valid and verifiable excuse.
4. Think carefully; formulate your answers, and then write them out concisely using English, logic, mathematics and pseudocode (no programming language syntax).

1. Algorithm Understanding (12 points)

Understand NAÏVE-STRING-MATCHER algorithm on p. 988 of the text. P and T are character arrays with the first character of P & T stored in array cells of index 1. The meaning of the condition "P[1..m]==T[s+1..s+m]" in step 4 is "compare P[1] with T[s+1], P[2] with T[s+2],...,compare P[m] with T[s+m] and if any of these comparisons returns FALSE then quit the character comparisons immediately and return FALSE otherwise continue and if all character comparisons succeed then return TRUE".

1a. If P=0001 and T=000010001010001, exactly how many character comparisons will the algorithm execute as a result of step 4 before it terminates?

31 2 points

1b. State all the values of s printed by the algorithm as a result of executing step 5 before it terminates:

s=1 2 points

s=5 2 points

s=11 2 points

1c. True or False? If |P|=m and all characters in P are the same character c_P, and |T|=n and all characters in T are the same character c_T, , m≤n, then if $c_P == c_T$ this represents a problem instance for which this algorithm will do the maximum number of character comparisons.

Circle one: True **False** 2 points

1d. True or False? If |P|=m and all characters in P are the same character c_P, and |T|=n and all characters in T are the same character c_T, , m≤n, then if $c_P != c_T$ this represents a problem instance for which this algorithm will do the minimum number of character comparisons.

Circle one: True **False** 2 points

2. Algorithm modification (12 points)

Replace the complex array equality condition in the **if** statement (step 4) of the NAÏVE-STRING-MATCHER with a **while** loop so that it behaves thus: “compare P[1] with T[s+1], P[2] with T[s+2],...,compare P[m] with T[s+m] and if any of these comparisons returns FALSE then quit the character comparisons immediately otherwise continue until all m characters of P have been compared”. Parts of the modified algorithm is given below. Fill in the blanks:

NAÏVE-STRING-MATCHER-MODIFIED(T: array [1..n] of char; P: array [1..m] of char, , m≤n)

```
1 n = T.length
2. m = P.length
3. for s = 0 to n–m
4.        j=1
5.        while j ≤ m 2 points  and P[j] 2 points  == T[s+j] 2 points
6.                j = j+1    2 points
7.        if j == m+1 then             2 points
8.                print “Pattern occurs with shift” s
```

When this algorithm’s execution reaches step 7 within any execution of the outer for loop, the value of the variable j carries some useful information. What is it? (circle one)

A. j = The number of character comparisons “P[j]==T[s+j]” that succeeded during the execution of the while loop

B. j = The number of character comparisons “P[j]==T[s+j]” that failed during the execution of the while loop

C. j = The number of characters of P that matched the substring T[s+1..s+m]

D. j = The number of characters of P that matched the substring T[s+1..s+m] + 1   2 points

E. j = The number of characters of P that matched the substring T[s+1..s+m] – 1

3. Algorithm Correctness (10 points)

A different modification of NAÏVE-STRING-MATCHER the effectively implements the same strategy is given below. But it is an incorrect algorithm.

MODIFIED-NAÏVE-STRING-MATCHER(T: array [1..n] of char; P: array [1..m] of char, , m≤n)

```
1 n = T.length
2 m = P.length
3 s = 0
4 while s<n–m+1 do
5         for i = 1 to m
6                 if P[i] != T[s+i] then
7                         s = s+i
8                         exit the i-loop and go to step 4
9         print "pattern occurs with shift" s
10        s = s+m
```

Multiple correct answers. Incorrectness stems from steps 7 & 10 that indicate a misguided attempt by the algorithm designer to shift the "window" by the number characters in T that matched with P in case of partial match (step 7) or full match (step 10).

Prove by Counterexample that it is incorrect by providing a problem instance for which it fails and explaining why it fails (complete the parts of the proof below).

Problem Instance: 2 points for P and 2 points for T if P & T are such that the algorithm above will produce an incorrect answer.

P=abc

T=aabcd

Correct answer or answers (correct values of shift s): 2 points for the correct shift values for P & T above

s=1

The value or values of s that the algorithm will print: 2 points for algorithm outputs for P & T above

nothing will be printed

Brief and precise explanation of why the algorithm prints incorrect answers for the given problem instance: 2 points for a correct explanation for algorithm failure

The algorithm fails on the above problem instance due to step 7. When P is being matched with T's substring T[1...3], the first character match will succeed (i=1 in the for loop) so steps 7 & 8 will not execute. The for loop will then execute with i=2, P[2]!=T[2] so this time step 7 will update s to s+i=s+2=2, so when the while loop executes next P will be matched with T's substring T[3...5]. Thus the algorithm will miss P's match with T's substring T[2...4].

4. Strategy & Algorithm Modification (5 points)

The strategy of the algorithm in Problem 1 is a "sliding window" strategy:

1. Match P[1..m] with a m-length substring of T[1..n] and if it succeeds print "Pattern occurs with shift" <the current value of s>=0
2. Then slide P one character to the right along T (i.e., s=s+1) and match P[1..m] with a m-length substring of T[s+1..m+1] and if it succeeds print "Pattern occurs with shift" <the current value of s>
3. Repeat step 2 until s=n–m and P[1..m] is matched with a m-length substring of T[n–m+1..n] and if this match succeeds print "Pattern occurs with shift" <the current value of s>= n–m

Now, if there are no repeated characters in P, i.e., all characters in P are distinct, it is possible to do the search for P in T faster. A modified strategy that does this is given below:

1. Use a variable *count* to keep track of the number of characters of P that match any substring of T. Start by matching P[1..m] with the first m-length substring of T, T[1..m].
2. If the match fails at the very first character of P, slide P one character to the right along T (i.e., update s to s+1) and then match P[1..m] with a m-length substring of T, T[s+1.. s+m].
3. If the match succeeds fully, print "Pattern occurs with shift" <the current value of s>.
4. If the match succeeds fully or partially, slide P *count* characters to the right along T (i.e., update s to s+*count*) and then match P[1..m] with a m-length substring of T, T[s+1.. s+m].
5. Repeat steps 2-4 until s>n–m. Assume m≥1 and m≤n.

Is this strategy correct? I.e., will it result in all the occurrences of P in T being correctly identified for all valid P=strings of at least one character in which all characters are distinct and T=strings of at least as many characters as there are in P? Circle one:

This strategy is correct **It is incorrect** 2 points

Explain your answer clearly and precisely in a few sentences:

3 points for any reasonable explanation even if it is not as detailed as the one below. 0 points if "It is incorrect" was chosen above.

This strategy exploits the fact that if no characters are repeated in P, then if first k characters of P matched with the first k characters of T's substring that P is being compared with, 1≤k≤m, then the "matching window" can be shifted to the right along T by k characters instead of by just one character (if no character of P matched, then the window should be shifted right by one character as in the original algorithm). This is because if P[1...k]==T[s+1...s+k] then P[1...k–1] cannot match with T[s+2...s+k] (because otherwise, i.e., P[1...k]==T[s+1...s+k] and P[1...k–1] == T[s+2...s+k], it implies that P[1]==P[2]==...==P[k] but all characters in P are distinct). Similarly P[1...k–2] cannot match with T[s+3...s+k] because if this were so, P[1]==P[3]==...==P[k] and this cannot be, P[1...k–3] cannot match with T[s+3...s+k], and so on. So the strategy correctly shifts s by 1 if there is no match and by *count* if there is a match of *count* characters, and prints s whenever there is a full match. Since now s can be incremented by more than 1, the iteration should stop once s crosses the value s=n–m, i.e., s>n–m.

5. Strategy to Algorithm (12 points)

The algorithm below implements the modified strategy above. It is incomplete. Fill in the blanks.

```
NAÏVE-STRING-MATCHER-FOR-DTSTINCT-PATTERN(T: array [1..n] of char; P: array [1..m] of char, m≤n)
n = T.length
m = P.length
s = 0
repeat
        x = 1
        while x ≤ m and P[x]==T[s+ x]
                x = x +1
        if x == m+1 then                    //P appears in T
                print "Pattern occurs with shift" s
        else if x == 1                      //The very first character of P Is not a match
                x = x + 1
        s = s + x –1
until s>n–m
```

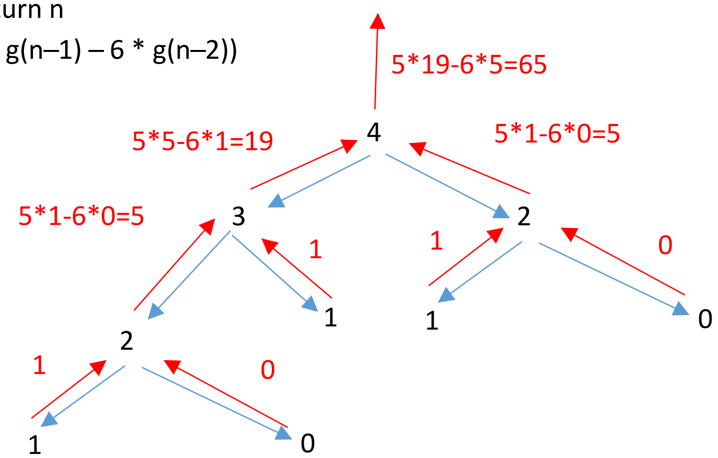
2 points for each filled blank

6. Understanding Recursive Algorithms (18 points)

Draw the recursion tree of the recursive algorithm when called with input n=4. Be sure to show all the input to each execution and the value returned by each execution in the tree.

g(n: non-negative integer)

1. if n ≤ 1 then return n
2. else return (5 * g(n–1) – 6 * g(n–2))



A thinking assignment: can you identify the mathematical function of n that this algorithm computes?

9 points for showing the input values of n at each of the 9 nodes of the Recursion Tree: 1 point each

9 points for showing each correct value returned by the 9 executions: 1 point each

7. Understanding Recursive Algorithms (5 points)

T: Binary Tree node; T.left and T.right: pointers to the left and right children of node T.

Mystery (T: Binary Tree Root Node)

1. if T.left == NULL and T.right == NULL then return 0
2. if T.left != NULL and T.right != NULL then
3.   return Larger(Mystery(T.left), Mystery(T.right)) + 1   //Larger(x,y) returns larger of x and y
4. if T.left != NULL then return Mystery(T.left)+1
5. if T.right != NULL then return Mystery(T.right)+1

What does Mystery compute? 5 points for any of the answers below

Binary tree height, or tree depth, or length of the longest path from tree root to a leaf, or number of edges on the path from the tree root to the deepest node.

8. Algorithm Design: Iterative (13 points)

An iterative strategy to move any and all zeroes in an array A of n numbers, n≥1, to the left end of the array:

1. Let leftp be a pointer to the leftmost index of A and rightp be a pointer to the rightmost index of A.
2. Move leftp right until leftp is pointing to a cell containing a non-zero number or leftp reaches the right end of A.
3. Move rightp left until rightp is pointing to a cell containing a zero or rightp reaches the left end of A.
4. If leftp and rightp are not equal or have not crossed (passed) each other, swap the numbers in cells pointed to by leftp and rightp and repeat 2-4.
5. If leftp and rightp are equal or have crossed (passed) each other then stop.

The corresponding iterative algorithm is given in part below. Complete it.

Move-zeroes-iterative(A: array [p..r] of number, r−p≥0)

1. leftp = p; rightp = r
2. repeat
3. swap(A[leftp], A[rightp])
4. while leftp≤r and A[leftp]==0 3 points
5. leftp=leftp+1 2 points
6. while rightp≥p and A[rightp]!=0 3 points
7. rightp= rightp−1 2 points
8. until leftp≥rightp 3 points

9. Algorithm Design: Recursive Divide & Conquer (13 points)

Turn the recursive divide & conquer strategy below to compute the total number of occurrences of a given character in a string of length zero or more represented as a character array into a recursive divide & conquer algorithm. The algorithm's header is provided; complete the rest.

"The number of occurrences of character X in string S is the sum of the number of occurrences of character X in the left half of string S and the number of occurrences of character X in the right half of string S"

This algorithm can be obtained by modifying the pseudocode of Merge Sort:

Character-Count-Recursive(S: array [p..r] of char, X: char)

1. if p<r then 2 points for applying divide & conquer only if S has more than 1 character
2. m=floor((p+r)/2) 1 point for calculating the index of the middle character
3. left-count= Character-Count-Recursive(S[p...m], X) 1 point for the 1st recursive call
4. right-count= Character-Count-Recursive(S[m+1...r], X) 1 point for the 2nd recursive call
5. return left-count+ right-count 2 points for returning the sum of the two counts
6. else if p==r then 2 points for dealing with 1-character string S
7. if S[p]==X then return 1 else return 0 2 points for returning the appropriate value
8. else return 0 2 points for dealing with 1-character string S