COMP 3270 Assignment 2 9 problems 100 points 10% Credit

Due before 11:59 PM Thursday September 16

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?

s = 0, 4 comparisons; s = 1, 4 comparisons (match); s = 2, 3 comparisons; s = 3, 2 comparisons; s = 4, 1 comparisons; s = 5, 4 comparisons (match); s = 6, 3 comparisons; s = 7, 2 comparisons; s = 8, 1 comparisons; s = 9, 2 comparisons; s = 10, 1 comparisons; s = 11, 4 comparisons (match) total comparisons: **31**

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

s = 1 **s = 5** **s = 11**

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

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. 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 and P[  j  ]==T[ s + j ]
6.          j = j + 1
7.    if j == m + 1 then
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

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 algorithms.

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
```

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:

P= [a, a, a, b]

T= [a, a, a, a, b]

Correct answer or answers (correct values of shift s):

s = 1

The value or values of s that the algorithm will print:

No values will be printed, which is incorrect

Brief and precise explanation of why the algorithm prints incorrect answers for the given problem instance:

When a match is not found the algorithm shifts past all the characters it has already compared, instead of shifting by one character. This results in no match being found, since it shifts past the match and prints nothing which is not the same as s = 1, the correct answer.

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..m] 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[2..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

Explain your answer clearly and precisely in a few sentences:

The main difference between how this algorithm works and the algorithm from problem 1, is the shifting. The first algorithm always shifts by one which means that is will check every possible substring of length m from T. The new algorithm, however, shifts over all the characters that matched with the substring from T. This works since all the characters in P are

distinct. So if certain characters of P match with T, then we know they will be the only character to match with that character of T, so we can skip over it after the match occurs. This will save time comparing characters but will only work if P consists of distinct characters.

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)

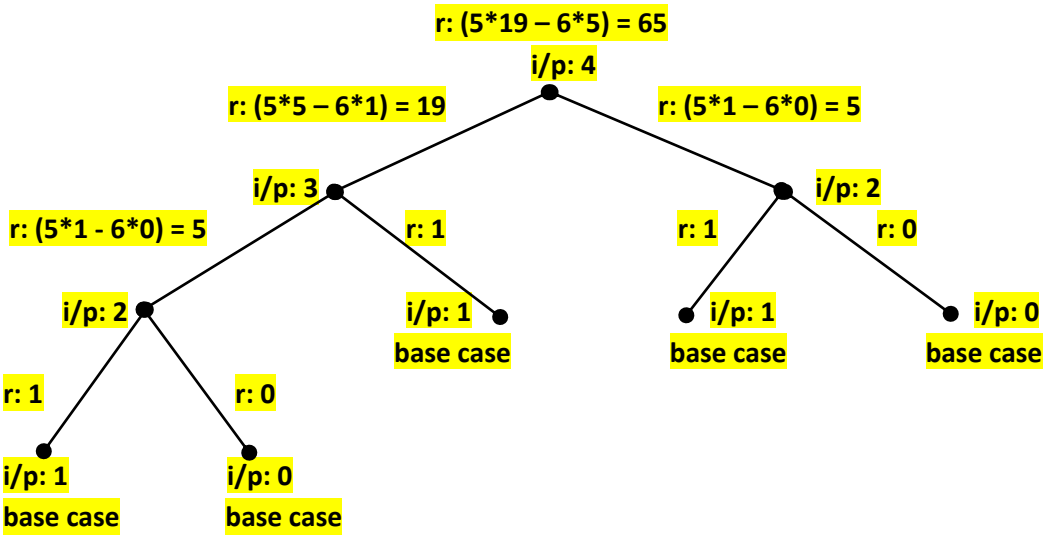
```
1 n = T.length
2. m = P.length
3. s = 0
4. repeat
5.    x = 1
6.    while x ≤ m and P[  x  ]==T[ s + x ]
7.          x = x + 1
8.    if x == m + 1 then          //P appears in T
9.          print "Pattern occurs with shift" s
10.   else if x == 1              //The very first character of P Is not a match
11.         x = x + 1
12.   s = s + x - 1
13. until s>n–m
```

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))



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?

The height of the binary tree: the longest path from the given root to the farthest descendent leaf

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 r 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**
5. **leftp = leftp + 1**
6. while rightp≥p and **A[rightp] != 0**
7. **rightp = rightp - 1**
8. until **leftp ≥ rightp**

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"

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

1. **if S.length == 1 and S[p] == X then**
2. **return 1**
3. **if S.length == 1 and S[p] != X then**
4. **return 0**
5. **mid = (p + r) / 2 // integer division**
6. **return character-count-recusive(S[p..mid], X) + character-count-recursive(S[mid+1..r], X)**