COMP 3270 Assignment 3 9 problems 150 points 10% Credit

Due before 11:59 PM Wednesday October 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. (8 points) Exercise 3.1-4 (pp. 53). Answer yes or no, and then **explain** why or why not. No credit without explanation.

Yes, $2^{n+1}=O(2^n)$ because $2^{n+1}=2*2^n \leq c*2^n$ where c=2, satisfying the Big-Oh definition for all $n \geq n_0=1$

0 if answer is no; 2 point for yes and 2 points for any reasonable explanation of why $2*2^n$ is $O(2^n)$; 2 points only if answer is yes but explanation is incorrect.

No, $2^{2n} \neq O(2^n)$ because if $2^{2n}=O(2^n)$ then $2^{2n}=2^n*2^n \leq c*2^n$ for all $n \geq n_0$ implying that there has to be a constant c such that $2^n \leq c$ $n \geq n_0$. But no constant can be greater than 2^n for all $n \geq n_0$!

0 if answer is yes; 2 points for no and 2 points for any reasonable explanation of why $2^{2n} \neq O(2^n)$; 2 points only if answer is no but explanation is incorrect.

2. (10 points)

```
Repeats (A[1…n]: integer array)
count = 0
number = −∞
for i = 1…(n-1) do
    if A[i]==A[i+1] then
        count=count+1
        if number ≠ A[i] then
            print number
    number = A[i]
print count
```

Does the algorithm above correctly solve the problem of counting the total number of all integers that repeat consecutively in the input array A correct? (E.g., if A contains 1 2 3 3 3 4 5 6 6 7 8 8 8 8 1 2 3 3 then count must be 11)? Answer yes or no. If your answer is yes, provide an explanation (no formal proof is required). If your answer is no, provide a counter example with input, correct answer, algorithm's output and a short explanation of why algorithm's output is different from the correct answer.

No, the algorithm is incorrect. 2 points

Proof by counterexample:

Input: A=[1 2 3 3 3 4 5 6 6 7 8 8 8 8 1 2 3 3] 2 points

Correct answer: 11 2 points; verify that this is the correct count for the input above.

Algorithm output: 7 2 points if this value is correct-answer – k where k = # of consecutive sets of repeating numbers in the input above. k=4 for this counterexample.

Explanation: The correct answer is the total count of consecutively repeated numbers, i.e., the count for 1 1 1 should be 3 but since step 4 checks A[i]==A[i+1] for i=1 to n-1, for every consecutive sets of 2 or more numbers it will miss counting the very first occurrence of that number (e.g., count will be 2 not 3 for 1 1 1). Thus the count for every consecutive set of repeating numbers will be one less, so if there are 4 sets of consecutive repeating numbers as in this counterexample, the cout will be 4 less than the correct answer. 2 points for any reasonable explanation.

0 points if the answer is that the algorithm is correct. Otherwise grade as above.

3. (20 points)

```
Select (A: Array [1..n] of distinct integers, k: integer between 1 and n)
1       for i=n down to (n–k+1)
2               position=i
3               for j=1 to (i–1)
4                       if A[j]>A[position] then position=j
5               if position≠i then
6                       temp=A[i]
7                       A[i]=A[position]
8                       A[position]=temp
9       print A[n-k+1]
```

The algorithm above correctly solves the problem of finding the k-th largest number in the input array. Complete the correctness proof by contradiction below by filling in the blanks.

Proof by contradiction:

1. Suppose the algorithm is **incorrect**.
2. That means that for at least one valid input array of n numbers and integer k, 1<=k<=n, (a) either it will not halt or (b) it will produce an incorrect answer.
3. Other than the two loops on lines 1 and 3, all other steps of the algorithm are basic operations that will halt. The loop statement in line 1 will exit after exactly (n–k+2) executions and the loop

statement in line 3 will execute i times for each execution of the outer loop in line 1. But since the value of i will always be between n and n–k+1, these will be finite executions also. So the algorithm must eventually halt.

4. So the only way the algorithm can be incorrect is if, for at least one valid input array of numbers of size n and integer k where 1<=k<=n, it does not print the k-th largest number.
5. Consider such an input. In the first execution of the outermost for loop, i=n, position=i=n initially, and the inner for loop will execute for j from 1 to (n–1).
6. Each time the inner loop is executed, A[j] is compared to A[position] and if the former is larger position is updated to j.
7. Therefore, after each execution of this inner loop, position must point to the index of the array cell that contains the larger value between A[j] and A[position].
8. Since this is repeated for every cell j=1…(n–1), and since position initially pointed to cell n, after the inner loop completes, position must point to the index of the largest number in the array A[1…n].
9. Then lines 5-8 will swap this largest number with the number in array cell n unless the largest number was already in A[i]=A[n]. Therefore, after the first execution of the outermost loop, the largest number from the array cells A[1]…A[n] will be in cell with index n.
10. In the second execution of the outermost for loop, i=(n–1), position=i=(n–1) initially, and the inner for loop will execute for j from 1 to (n–2). So by a similar argument, after the second execution of the outermost loop, the largest number in the array cells A[1]…A[n–1] will be in cell with index (n–1).
11. Since the first largest number in array cells 1…n was already in cell n before the second execution of the outer loop started, this means that the second largest number in the array will now be in cell (n–1).
12. By a similar argument, after the k-th execution of the outermost loop, the k-th largest number in the array will be in cell with index (n–k+1).
13. Line 9 prints this number. So for any array of numbers of size n and integer k where 1<=k<=n, the algorithm will print the k-th largest number in the array.
14. This contradicts step 4 of the proof.
15. So our assumption in step 1 must be wrong, i.e. the algorithm has to be correct.

Each filled blank carries 1 point. As long as the meaning is correct, words and notations different from above are acceptable, e.g., “A[1]…A[n]” or “A[1...n]” or “array indexes 1…n” are all equivalent.

4. (18 points) Determine a Loop Invariant for the outer for loop of the Selection-Sort algorithm below that allows you to prove that the algorithm is correct. Then state the proof. Parts of both are given – fill in the blanks. Understanding how this algorithm works will give you the needed information to construct an appropriate loop invariant.

```
Selection-Sort (A: Array [1..n] of integer)
1    for i=n down to 2
2        position=i
3        for j=1 to (i–1)
4            if A[j]>A[position] then position=j
```

```
5           if position≠i then
6               temp=A[i]
7               A[i]=A[position]
8               A[position]=temp
```

Explanatory Notes: The key here is to first understand how Selection Sort sorts – by finding the position of the largest number in the array A[1,…n] and swapping that with the nth cell of the array, then finding the position of the largest number in the array A[1,…(n-1)] and swapping that with the (n-1)th cell of the array, and repeating this process for subarrays A[1,…(n-2)], A[1,…(n-3)],…,A[1,2]. Note that the outer loop is executed (n-1) times, with the value of the loop variable i going from n down to 2. So during the first execution of the outer loop (line 1), i=n, after which A[n] will contain the (first) largest number in the array and i will be decremented to n-1. During the second execution of the outer loop, i=n-1, after which A[n] will contain the first largest number in the array and A[n-1] will have the second largest number in the array and i will be decremented to n-2. During the third execution of the outer loop, i=n-2, after which A[n] will contain the first largest number in the array and A[n-1] will have the second largest number in the array, and A[n-2] will have the third largest number in the array and i will be decremented to n-3. During the kth execution of the outer loop (we are using k to count the loop executions because the loop variable i is decremented and so does not count the number of executions of the loop itself), i=n-k+1, after which A[n] will contain the first largest number in the array and A[n-1] will have the second largest number in the array,…,and A[n-k+1] will have the kth largest number in the array and i will be decremented to n-k. Now state this in such a way that allows you to prove that by doing this repeatedly in a loop, at the end of the loop all numbers in the array will be in their sorted place.

Loop Invariant:
Before kth execution of the outer loop (line 1), the loop variable i=n–k+1, and the (k–1)th, (k–2)th,…,2nd, and 1st largest numbers in array A will be in the subarray A[(n–k+2)…n].
Initialization:
The LI should hold before the first execution of the loop: Before the 1st execution of the loop, the loop variable i=n-1+1=n, and 0th,…,1st largest numbers in array A will be in subarray A[(n+1)…n]. But both the 0th largest number and the subarray A[n+1…n] are undefined for an array of n numbers. So the LI trivially holds.
Maintenance:
Here we have to show that if the LI held true before the kth execution of the loop, it must also be true after the kth execution, i.e., before the (k+1)th execution.
So suppose before the kth execution of the loop the LI holds, i.e., the loop variable i=n–k+1, and the (k–1)th, (k–2)th,…,2nd, 1st largest numbers in array A are in the subarray A[(n–k+2)…n].
During this execution, the local variable position is initialized to i=n–k+1 (line 2).
The for loop executes for each value of j from 1 to (i–1)=(n–k) (line 3).
Each time, if a number A[j] greater than A[position] is found then position is updated to j. Therefore, since position starts out being (n–k+1) and j goes from 1 to (n–k), at the end of this loop, position will contain the index of the largest number in the subarray A[1…(n–k+1)].

Line 5 checks to see if this index is the same as i=(n–k+1). If it is, then nothing is done, and the largest number in A[1,…(n–k+1) is in array cell A[n–k+1]. If it is not, then A[position] and A[n-k+1] are swapped by lines 6-8 so that the largest number in A[1,…(n–k+1) is in A[n–k+1].
Since the (k–1)th, (k–2)th,…,2nd, 1st largest numbers in array A were already in the subarray A[(n–k+2)…n] before the kth execution of the loop started, this means that by the end of the kth execution of the loop, the kth largest number will be in array cell A[n–k+1].
Finally, by the nature of the for loop, the loop variable i is decremented to n–k, and the kth execution of the loop finishes.
So after the kth execution of the loop finishes, i.e., before the (k+1)th execution of the loop, the loop variable i=n–k, and the kth, (k–1)th, (k–2)th,…,2nd, 1st largest numbers in array A will be in the subarray A[(n–k+1)…n].
Thus, if the LI holds before the kth execution of the loop, it will hold before the (k+1)th execution of the loop.
Termination:
As we have proved initialization and maintenance, it holds that LI must be true after the loop finishes. The loop is executed (n–1) times. So after the (n–1)th execution, i.e., before the nth execution, the loop variable i=n–n+1=1, and the (n–1)th, (n–2)th,…,2nd, 1st largest numbers in array A will be in the subarray A[(n–n+2)…n], i.e., A[2,…,n]. Since the first (n-1) largest numbers are thus now already sorted, what is in A[1] must be the nth largest, i.e., the smallest, number in the input array. Therefore A is fully sorted in the increasing order.
Each filled blank carries 1 point. As long as the meaning is correct, words and notations different from above are acceptable.

5. (24 points) Calculate the complexity T(n) of the Bubble-sort algorithm below. Calculate the constant cost of a step by assuming that **each** basic operation included in that step – addition, subtraction, multiplication, division, array-read, array-write, assigning a value to a variable, returning a value, etc. – has a cost of 1. So the cost of executing a statement once is to be calculated as the total number of basic operations that have to be executed. Fill in the table below, then determine the expression for T(n) and simplify it to produce a polynomial in n. In the second column for steps 4-9, provide sigma (summation) notation.

```
Bubble-sort (A: Array [1..n] of integer)
1       i=1
2       while i≤(n–1)
3               j=1
4               while j≤(n–i)
5                       if A[j]>A[j+1] then
6                               temp=A[j]
7                               A[j]=A[j+1]
8                               A[j+1]=temp
9                       j=j+1
10              i=i+1
```

Step	Cost of each execution	Total # of times executed
1	1	1
2	5	n
3	1	n-1 OR $\sum_{i=1}^{i=n-1} 1$
4	6	$\sum_{i=1}^{i=n-1}(n-i+1)$
5	8	$\sum_{i=1}^{i=n-1}(n-i)$
6	4	$\sum_{i=1}^{i=n-1}(n-i)$
7	7	$\sum_{i=1}^{i=n-1}(n-i)$
8	5	$\sum_{i=1}^{i=n-1}(n-i)$
9	3	$\sum_{i=1}^{i=n-1}(n-i)$
10	3	n-1

Each table entry caries 1 point. No grade penalty if the constants in column 1 are off by +/– 3. Total 20 points.

T(n) for Bubble-sort = 1+5n+(n-1)+6$\sum_{i=1}^{i=n-1}(n-i+1)$+(8+4+7+5+3)($\sum_{i=1}^{i=n-1}(n-i)$)+3(n-1) = 16.5n^2-1.5n-9

3 points for T(n): 1 point for a n^2-term, 1 point for a n-term, and 1 point for a constant term; no penalty for the constant coefficients being different from the above.

1 point for attempting this question, for a total of 24 points.

6. (7 points) Exercise 2.3-4 (pp. 39). You need to first provide the recursive algorithm (parts of it given below; complete it), using pseudocode conventions, and then develop and state the two recurrence relations. You do not have to provide the constant values; instead use Θ(1) for a constant value, Θ(n) for a constant multiplier of n, etc. You do not need to solve the recurrences.

```
Insertion-sort-recursive(A[1..n])
if n>1 then
    Insertion-sort-recursive(A[1..n–1])
    key=A[n]
    i=n–1
    while i>0 and A[i]>key
        A[i+1]=A[i]
        i=i-1
```

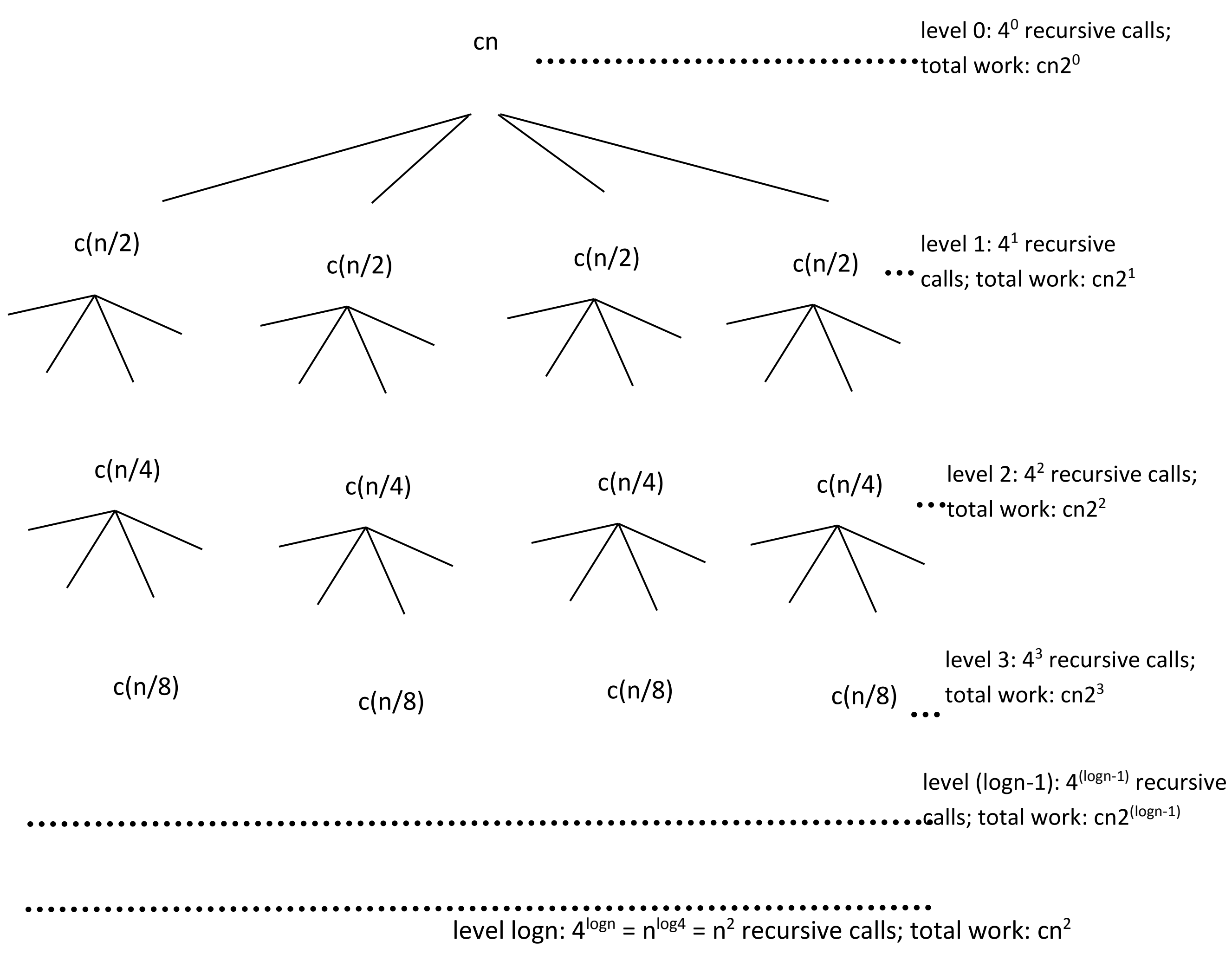
A[i+1]←key

T(n)= ______Θ(1)____ if n=1; T(n) = T(__n–1___) + _____Θ(n)__ if n> 1

1 point for each filled blank.

7. (25 points) T(n)=4T(n/2)+cn and T(1)=c for a recursive algorithm. Determine the polynomial T(n) for this recursive algorithm using the Recursion Tree Method. (a) Fill in the table (drawing the tree may help you fill in the table entries). (b) Then add total work done across levels and simplify to produce a polynomial expression for T(n). (3) State the Θ complexity of the algorithm. You will need to use the following result, where a and b are constants and a summation simplification result from Appendix A of the text.

$$a^{\log_b n} = n^{\log_b a}$$


cn
level 0: 4^0 recursive calls; total work: $cn2^0$
c(n/2)
c(n/2)
c(n/2)
c(n/2)
level 1: 4^1 recursive calls; total work: $cn2^1$
c(n/4)
c(n/4)
c(n/4)
c(n/4)
level 2: 4^2 recursive calls; total work: $cn2^2$
c(n/8)
c(n/8)
c(n/8)
c(n/8)
level 3: 4^3 recursive calls; total work: $cn2^3$
level (logn-1): $4^{(\log n-1)}$ recursive calls; total work: $cn2^{(\log n-1)}$
level logn: $4^{\log n} = n^{\log 4} = n^2$ recursive calls; total work: cn^2

The tree is provided for explanatory purposes. Grade the table as follows:

level	Level number	Total # of recursive executions at this level	Input size to each recursive execution	Work done by each recursive execution, excluding the recursive calls	Total work done by the algorithm at this level
0	0	4^0	n	cn	cn
1	1	4^1	n/2	c(n/2)	2cn
2	2	4^2	n/4	c(n/4)	4cn
The level just above the base case level	(logn-1)	$4^{(\log n-1)}$	$n/2^{(\log n-1)}=2$	$c(n/2^{(\log n-1)})=2c$	$2^{(\log n-1)}cn$
Base case level	logn	$4^{\log n} = n^{\log 4} = n^2$	$n/2^{\log n}=1$	c	$2^{\log n}cn=n^{\log 2}cn= cn^2$

1 point for each table entry – total 22 points

$T(n) = 2cn^2 – cn$ 2 points: 1 for the $2cn^2$ term and 1 for the cn term.

Complexity order of the algorithm = Θ(_n^2_) 1 point

Explanation: Total work = $cn2^0 + cn2^1 + cn2^2 + cn2^3 + … + cn2^{(\log n-1)} + cn^2 = cn\Sigma_{i=0}^{i=(\log n-1)}2^i + cn^2$ = {using the summation simplification result from Appendix A of the text $\Sigma_{k=0}^{k=n}x^k = (x^{n+1}–1)/(x–1)$} $cn(2^{\log n}–1) + cn^2 = cn^2 – cn + cn^2 = 2cn^2 – cn = T(n)$. As T(n) is a polynomial in n of degree 2, the tightest bound or best complexity order of $T(n)=\Theta(n^2)$.

8. (20 points). Do Exercise 4.5-1 (pp 96).

(a)	a=__2__	b=____4__	f(n)=__$1=n^0$___	Which case applies? ___1____	$T(n)=\Theta(\sqrt{n}=n^{0.5})$
(b)	a=___2___	b=____4_____	f(n)=__$\sqrt{n}=n^{0.5}$_	Which case applies? ___2__	$T(n)=\Theta(\sqrt{n}\lg n=n^{0.5}\lg n)$
(c)	a=___2___	b=____4_____	f(n)=____n^1____	Which case applies? ___3____	$T(n)=\Theta(n)$
(d)	a=__2___	b=____4_____	f(n)=____n^2____	Which case applies? ___3____	$T(n)=\Theta(n^2)$

1 point per filled blank

9. (18 points) Solve the recurrences T(n)=T(n-1)+cn; T(1)=c where c is a constant, first by the backward substitution method and then by the forward substitution method. In each case fill in the blanks.

Backward substitution method:

T(n) =T(n-1) + nc

=T(n–2) + (n–1)c + nc

= T(n–3) + (n–2)c + (n–1)c + nc

…

$=T(1)+2c+3c+4c+\ldots+nc = c+2c+3c+4c+\ldots+nc = c(1+2+3+\ldots+n) = cn(n+1)/2 = (cn^2+cn)/2$

$=\Theta(n^2)$ 2 points

Forward substitution method:

$T(1)=c$

$T(2)=T(1)+2c = c + 2c$

$T(3)=T(2)+3c = c + 2c + 3c$

…

$T(n) = c+2c+3c+4c+\ldots+nc = c(1+2+3+\ldots+n) = cn(n+1)/2 = (cn^2+cn)/2$

$= \Theta(n^2)$ 2 points

1 point for each filled blank except as noted above.