COMP 3270 Assignment 3 9 problems 150 points 10% Credit
Due before 11:59 PM Thursday October 14

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.

Is $2^{n+1} = O(2^n)$? Yes

Let $2^{n+1} = T(n)$
It must be shown that $cg(n) \geq T(n)$ where c is a constant and $g(n) = 2^n$
2^{n+1} can be written as $2^n * 2$
For the question to be true it must be true that $c(2^n) \geq 2^{n+1}$
Since $2^{n+1} = 2 * 2^n$ then it is also true that $2(2^n) \geq 2^{n+1}$
2 is just a constant with $g(n) = 2^n$ therefore the Big-Oh is $O(g(n)) = O(2^n)$

Is $2^{2n} = O(2^n)$? No

Let $2^{2n} = T(n)$
It must be shown that $cg(n) \geq T(n)$ where c is a constant and $g(n) = 2^n$
2^{2n} can be written as $2^n * 2^n$
For the question to be true it must be true that $c(2^n) \geq 2^{2n}$. However, there is no constant c where this inequality can become true. Therefore $2^{2n} \neq O(2^n)$?

2. (10 points)

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

```
6                       if number ≠ A[i] then
7                               print number
8               number = A[i]
9       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
Problem Instance: A is [1, 2, 3, 4, 4]
The correct output is 2
The algorithm outputs 1, which is incorrect
The output is incorrect because when the algorithm finds repeating consecutive integers it counts wrong. In this instance it saw that 4 and 4 were two repeating consecutive integers but it only increased the count to 1.

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 **not print the k-th largest number** .
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 **k + 1** 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 with the largest integer looked at by the algorithm up to this point**.
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 with the largest integer in the array**.
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] to A[n]** will be in cell with index **n**.
10. In the second execution of the outermost for loop, i= **n – 1**, position= **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] to A[n – 1]** will be in cell with index **A[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 **2 and 4** of the proof.
15. So our assumption in step 1 must be wrong, i.e. the algorithm has to be correct.

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 this algorithm sorts – by finding the position of the largest number in the array A[1,…n] and swapping that with the n^{th} 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 k^{th} 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, etc., and i will be decremented to n-k.

Loop Invariant:
Before k^{th} execution of the outer loop (line 1), the loop variable i=n-k+1, and **the (k-1)** largest numbers in array A will be in the subarray **A[i+1…n]**

Initialization:
The LI should hold before the first execution of the loop: Before the 1^{st} execution of the loop, the loop variable i=n-1+1=n, and 0^{th},…,1^{st} largest numbers in array A will be in subarray A[(n+1)…n]. But both the 0^{th} 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 k^{th} execution of the loop, it must also be true after the k^{th} execution, i.e., before the $(k+1)^{th}$ execution.
So suppose before the k^{th} execution of the loop the LI holds, i.e., the loop variable i=n-k+1, and the **k-1** largest numbers in array A are in the subarray **A[i+1…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 subarray **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 subarray **A[1…n-k+1]** is in array cell **A[1…n-k+1]**

Since the **k-1** largest numbers in array A were already in the subarray **[i+1…n]** before the k^{th} execution of the loop started, this means that by the end of the k^{th} execution of the loop, the k^{th} 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 k^{th} execution of the loop finishes.
So after the k^{th} execution of the loop finishes, i.e., before the $(k+1)^{th}$ execution of the loop, the loop variable i= **n-k**, and the **k** largest numbers in array A will be in the subarray **A[n-k+1…n]**.
Thus, if the LI holds before the k^{th} 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 n^{th} execution, the loop variable i= **1**, and the **n-1** largest numbers in array A will be in the subarray **A[2…n]**. Since the first (n-1) largest numbers are thus now already sorted, what is in A[1] must be the n^{th} largest, i.e., the smallest, number in the input array. Therefore A is fully sorted in the increasing order.

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**

4	**6**	$\sum_{i=1}^{n-1} t_i$
5	**8**	$\sum_{i=1}^{n-1} t_i - 1$
6	**4**	$\sum_{i=1}^{n-1} t_i - 1$
7	**7**	$\sum_{i=1}^{n-1} t_i - 1$
8	**5**	$\sum_{i=1}^{n-1} t_i - 1$
9	**3**	$\sum_{i=1}^{n-1} t_i - 1$
10	**3**	**n – 1**

t_i = the number of times the while loop test in line 4 is executed for the value of i

T(n) for Bubble-sort = **1(1) + 5(n) + 1(n – 1) + 6(** $\sum_{i=1}^{n-1} t_i$ **) + 8(** $\sum_{i=1}^{n-1} t_i - 1$ **) + 4(** $\sum_{i=1}^{n-1} t_i - 1$ **) + 7(** $\sum_{i=1}^{n-1} t_i - 1$ **) + 5(** $\sum_{i=1}^{n-1} t_i - 1$ **) + 3(** $\sum_{i=1}^{n-1} t_i - 1$ **) + 3(n – 1)**

= 1 + 5n + n – 1 + 6(n(n – 1)/2) + 8((n(n – 1)/2) – n + 1) + 4((n(n – 1)/2) – n + 1) + 7((n(n – 1)/2) – n + 1) + 5((n(n – 1)/2) – n + 1) + 3((n(n – 1)/2) – n + 1) + 3n – 3

= $\frac{3(11n^2-23n+16)}{2}$

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 – 1]**
    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

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.

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

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	$\mathbf{n}$	$\mathbf{1}$	$\mathbf{cn}$	$\mathbf{cn}$
1	1	$\frac{\mathbf{n}}{\mathbf{2}}$	$\mathbf{4}$	$\frac{\mathbf{cn}}{\mathbf{2}}$	$\frac{\mathbf{4cn}}{\mathbf{2}} = \mathbf{2cn}$
2	2	$\frac{\mathbf{n}}{\mathbf{4}}$	$\mathbf{16}$	$\frac{\mathbf{cn}}{\mathbf{4}}$	$\frac{\mathbf{16cn}}{\mathbf{4}} = \mathbf{4cn}$
The level just above the base case level	$\mathbf{\log_2 n - 1}$	$\mathbf{2}$	$4^{(\log_2 n)-1} = \frac{\mathbf{n^2}}{\mathbf{4}}$	$\frac{\mathbf{cn}}{\mathbf{2^{(\log_2 n)-1}}} = \mathbf{2c}$	$\frac{\mathbf{4^{(\log_2 n)-1}cn}}{\mathbf{2^{(\log_2 n)-1}}} = \frac{\mathbf{n^2 c}}{\mathbf{2}}$
Base case level	$\mathbf{\log_2 n}$	$\mathbf{1}$	$\mathbf{4^{\log_2 n} = n^2}$	$\mathbf{c}$	$\mathbf{cn^2}$

T(n) = $\sum_{k=0}^{\log_2 n} 4^k \cdot \frac{cn}{2^k} = cn \cdot \sum_{k=0}^{\log_2 n} \frac{4^k}{2^k} = cn \cdot \sum_{k=0}^{\log_2 n} 2^k = cn \cdot (2n-1) = 2cn^2 - cn$

Complexity order of the algorithm = Θ($\mathbf{n^2}$)

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

(a) a= **2** b= **4** f(n)= **1** Which case applies? **1** T(n)=Θ($\mathbf{\sqrt{n}}$)

(b) a= **2** b= **4** f(n)= $\mathbf{\sqrt{n}}$ Which case applies? **2** T(n)=Θ($\mathbf{\sqrt{n}\log(n)}$)

(c) a= **2** b= **4** f(n)= **n** Which case applies? **3** T(n)=Θ(**n**)

(d) a= **2** b= **4** f(n)= $\mathbf{n^2}$ Which case applies? **3** T(n)=Θ($\mathbf{n^2}$)

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**

…

= $\frac{\mathbf{c\,(n(n+1))}}{\mathbf{2}}$ //the final expression without a T() term

=Θ(**n^2**)

Forward substitution method:

T(1)= c

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

T(3)= **T(1)** + **2c** + **3c**

…

= $\frac{\mathbf{c\,(n(n+1))}}{\mathbf{2}}$ //the final expression without a T() term

=Θ(**n^2**)