**COMP 3270**

**Assignment 2**

**100 points**

**Due Friday, June 16th by 11:59PM**

Instructions:

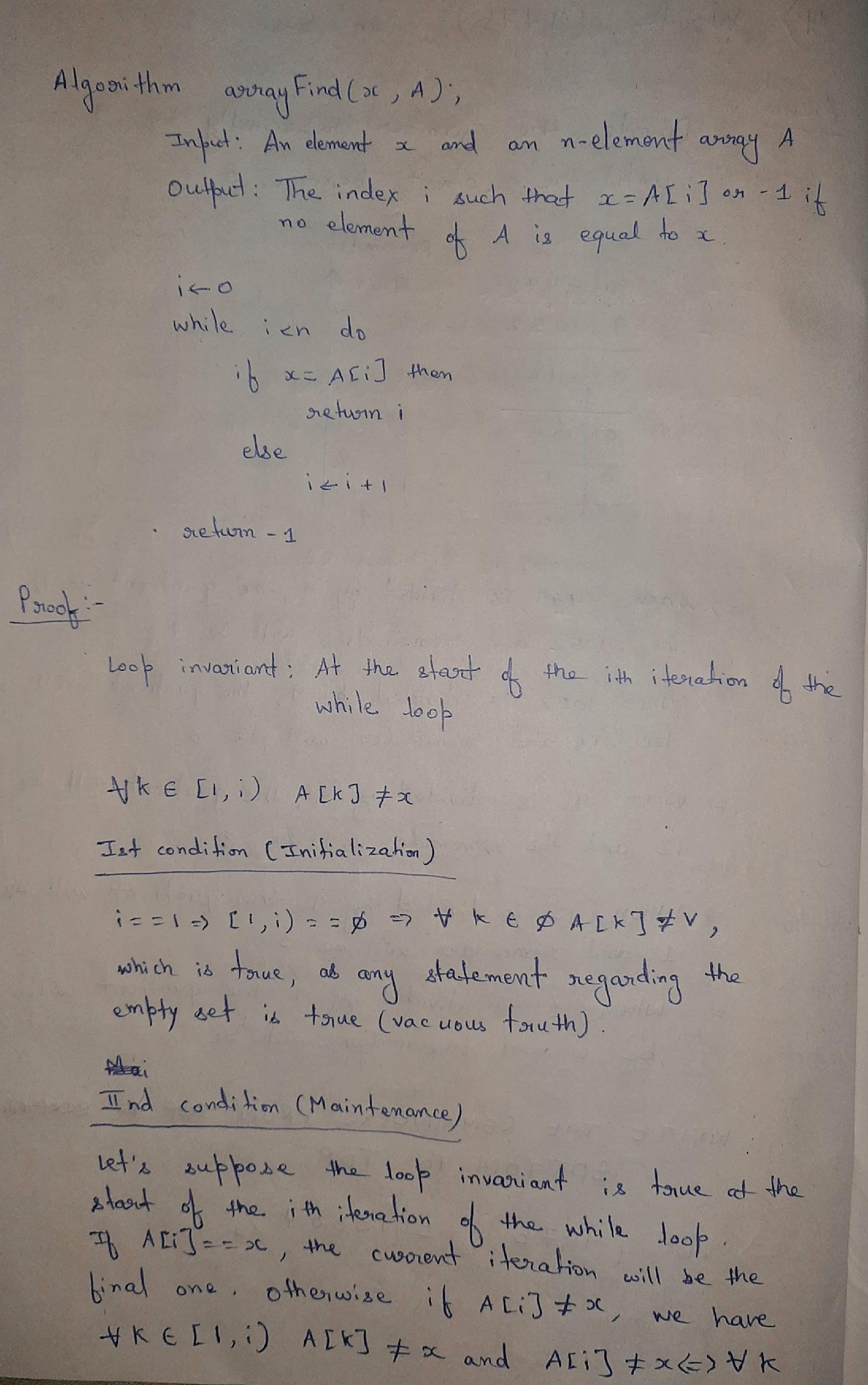
1. This is an individual assignment. There are 8 problems.
2. Late submissions **will not** be accepted unless prior permission has been granted or there is a valid and verifiable excuse.
3. Think carefully; formulate your answers, and then write them out concisely using English, logic, mathematics and pseudocode (no programming language syntax).
4. Type your final answers in this Word document.
5. Don’t turn in handwritten answers with scribbling, cross-outs, erasures, etc. If an answer is unreadable, it will earn zero points.
6. **(6 points)** Prove that the following algorithm is correct by using the “Proof by Loop Invariants” method. **Hint**: Loop Invariant **Si=x is not equal to any of the first i elements of the array**

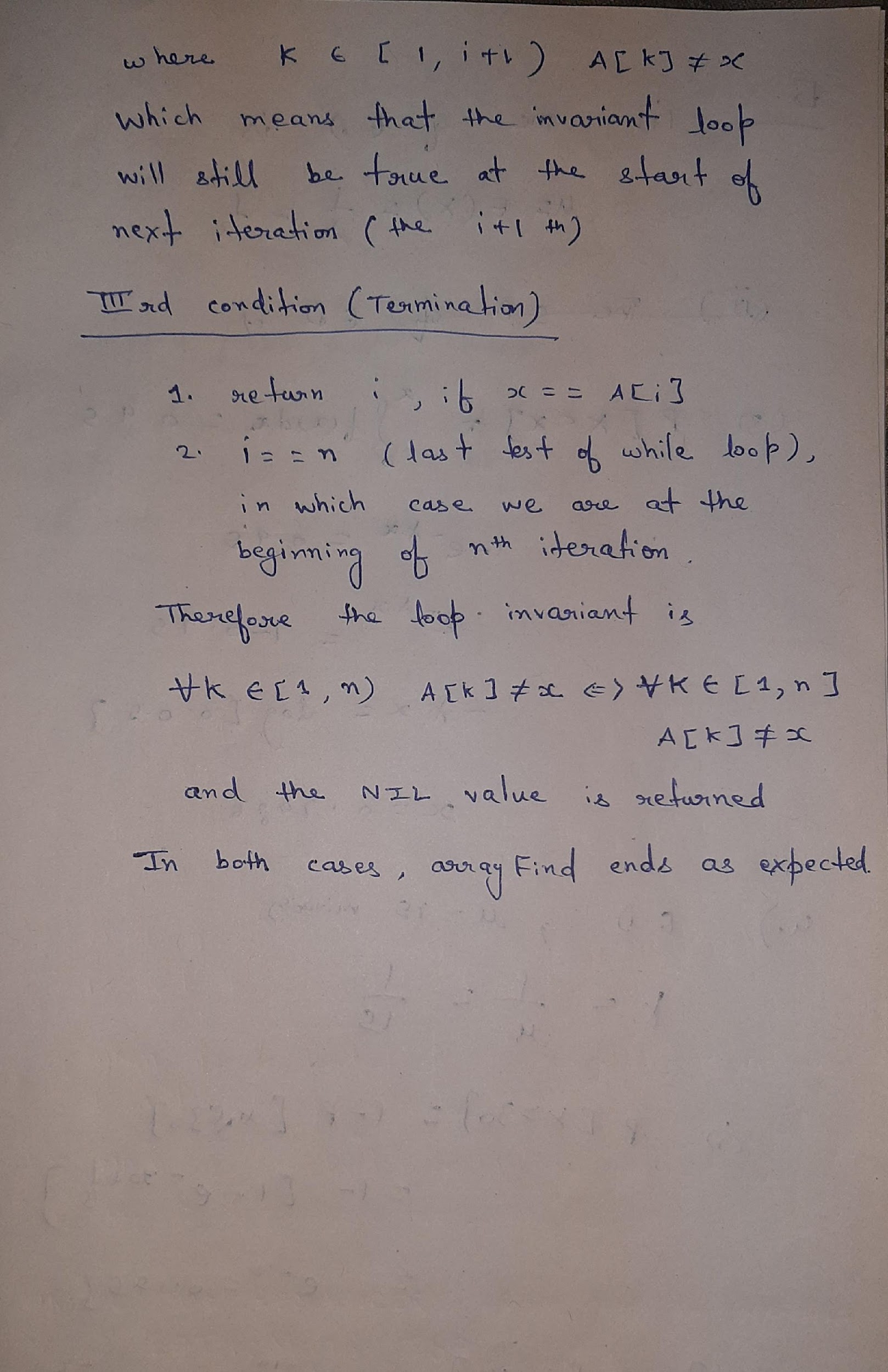
Text, letter

Description automatically generated

Loop invariant is used to prove correctness of any algorithm.  
We must satisfy three conditions to prove the corectness of any algorithm through loop invariant.  
1. Initialization: It is a statement which is true before first iteration.  
2. Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.  
3. Termination: Whenever the loop terminates, the invariant gives us a useful property which helps in showing the correctness of that algorithm.

Proof of arrayFind algorithm through loop invariant:





**2. (5 points)** Order the following list of functions by the big-Oh notation. Group together (for example by underlining) those functions that are big-Theta to each other.

Text

Description automatically generated with medium confidence

1. 1/*n*
2. 2100
3. log log *n*
4. sqrt(log *n*) (Note: too see this is greater than log log *n*, consid er substituting *x* = log *n* into both formulae)
5. log2*n*
6. *n*0.01
7. sqrt(*n*), 3 *n*0.5
8. 2log *n*, 5 *n*
9. *n* log4*n*, 6 *n* log *n*
10. 2 *n* log2 *n*
11. 4 *n*3/2
12. 4log *n*
13. *n*2 log *n*
14. *n*3
15. 2*n*
16. 4*n*
17. 22*n*

**3. (11 points)** Describe a method for finding both the minimum and maximum of n numbers using fewer than 3n/2 comparisons. ***Hint:*** First construct a group of candidate minimums and a group of candidate maximums. **You are required to use an inductive strategy.**

Let us consider an array A of n elements. We declare two variables MX and MN representing maximum and minimum elements. We have 2 cases.

If n is odd, then we initialise MN and MX elements as A[0], else we initialise MX and MN as the maximum and minimum of A[0] and A[1] respectively.

Starting with the next index in each case, we iterate through the arrays with increments of two. We basically check in pairs. If A[i] is greater than A[i+1], then A[i] is a candidate for maximum and A[i+1] is a candidate for minimum. Hence we check for these respectively. This takes 3 comparisons whatever be the case. Similar is the case when A[i] is less than A[i+1], here A[i] is a candidate for minimum and A[i+1] is a candidate for maximum.

As each step takes 3 comparisons and we iterate through the array in increments of 2, the total number of comparisons is less than 3\*(n/2).

Infact, the exact number of comparisons are as follows:  
If n is odd: 3\*(n-1)/2

If n is even: (3n/2) -2

**4. (12 points)**

Algorithm Mystery(A: Array [i..j] of integer) i & j are array starting and ending indexes

if i=j then return A[i]

else

k=i+floor((j-i)/2)

temp1= Mystery(A[i..k])

temp2= Mystery(A[(k+1)..j]

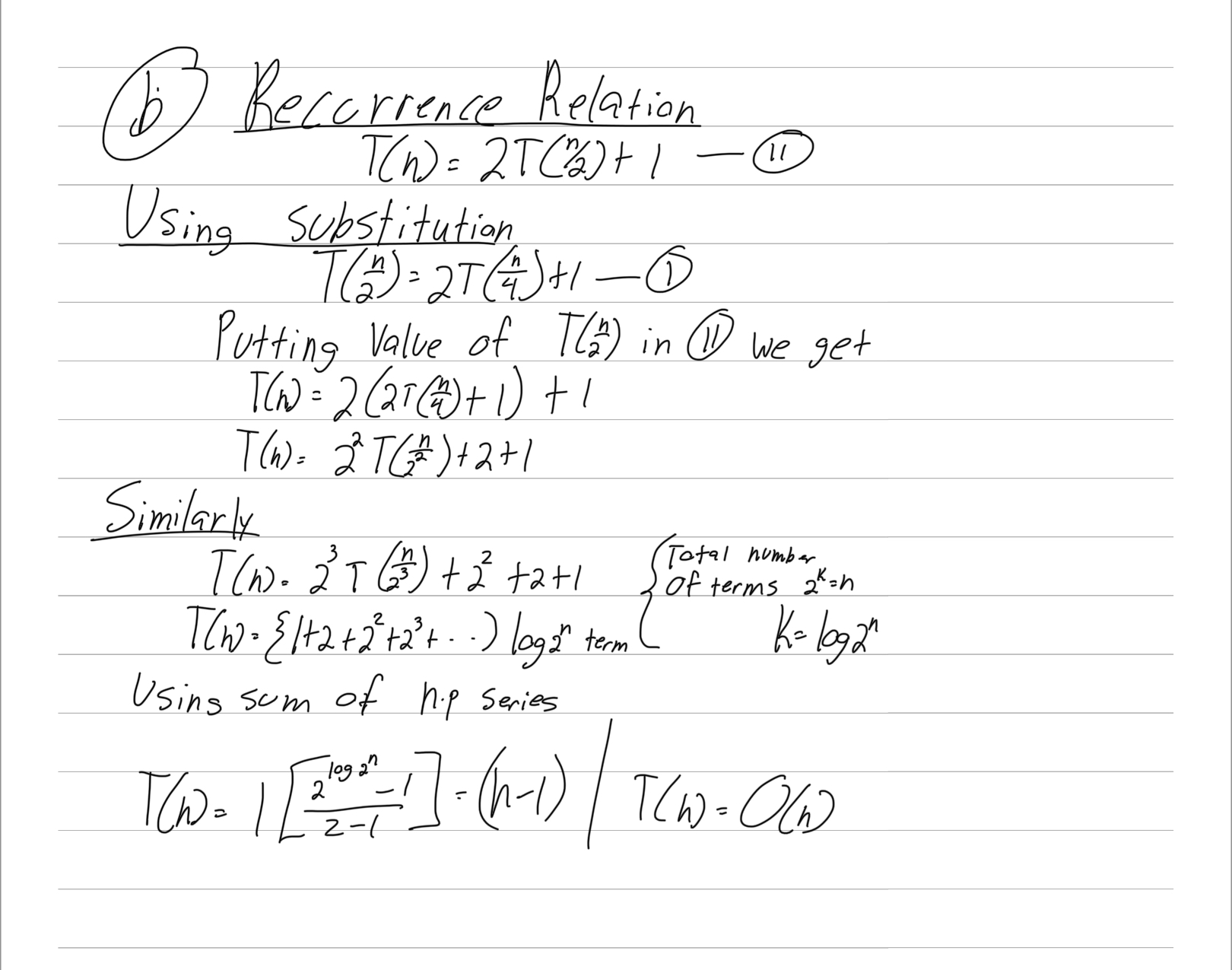
if temp1<temp2 then return temp1 else return temp2

(a) (1 points) What does the recursive algorithm above compute?

The given recurisve algorithm compute the minimum element in array A[1..j]

G+ find min value element

(b) (4 points) Develop and state the two recurrence relations exactly (i.e., determine all constants) of this algorithm by following the steps outlined in L7-Chapter4.ppt. Determine the values of constant costs of steps using directions provided in L5-Complexity.ppt. Show details of your work if you want to get partial credit.



(c) (6 points) Use the Recursion Tree Method to determine the precise mathematical expression T(n) for this algorithm. First, simplify the recurrences from part (b) by substituting the constant “c” for all constant terms. Drawing the recursion tree may help but you do not have to show the tree in your answer; instead, fill the table below. Use the examples worked out in class for guidance. Show details of your work if you want to get partial credit.

You will need the following result:



| 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 |
| --- | --- | --- | --- | --- | --- |
| Root | 0 | 1 | n | 1 | 1 \* 1 = 1 |
| One level below root | 1 | 2 | n/2 | 1 | 1 \* 2 = 2 |
| Two levels below root | 2 | 4 | n/4 | 1 | 1 \*4 = 4 |
| The level just above the base case level | Log 2 ^n - 1 | 2^(log 2^n –1) | 2 | 1 | 1 \*2^ (log 2^n –1) |
| Base case level | Log 2^n | n | 1 | 1 | n |

(d) (1 points) Based on T(n) that you derived, state the order of complexity of this algorithm:

T(n) = (1 +2 +4+8+…) log 2 ^n times

T(n) = O(n)

**5. (20 points)** T(n)=7T(n/8)+cn; T(1)=c. Determine the polynomial T(n) for the recursive algorithm characterized by these two recurrence relations, using the Recursion Tree Method. Drawing the recursion tree may help but you do not have to show the tree in your answer; instead, fill the table below. You will need to use the following results, where and b are constants and x<1:



| 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 at this level |
| --- | --- | --- | --- | --- | --- |
| Root | 0 | 1 | n | cn | C\*n |
| 1 level below | 1 | 7 | n/8 | C\*n/8 | C\*7/8\*n |
| 2 levels below | 2 | 7^2 | n/8^2 | C\*n/8^2 | C\* 7^2/8^2\*n |
| The level just above the base case level | Log8n-1 | 7log8n/7 | 8 | C\*8 | C\*8/7\* 7^ log8n |
| Base case level | log8n | 7^ og8n | 1 | c | C\*7^ log8n |

T(n) = ***T(n) = Θ(n)***

**6. (11 points)** Use the substitution method to prove the guess that is indeed correct when T(n) is defined by the following recurrence relations: T(n)=3T(n/3)+5; T(1)=5. At the end of your proof state the value of constant c that is needed to make the proof work.



Statement of what you have to prove:

That T(1) = O(n)

Base Case proof:

T(1)=O(n)

As T(1)=5 hence T(1) = O(n)

Inductive Hypotheses:

T(k) = O(k) for some fixed c and N for all n >N Ii.e T(k) <ck

Inductive Step:

T(k+1) = O(k+1)

Using induction

T9kh) < 3c k/3 +5

T(k+1) < ck +5

< c(k+1) + 5 –c

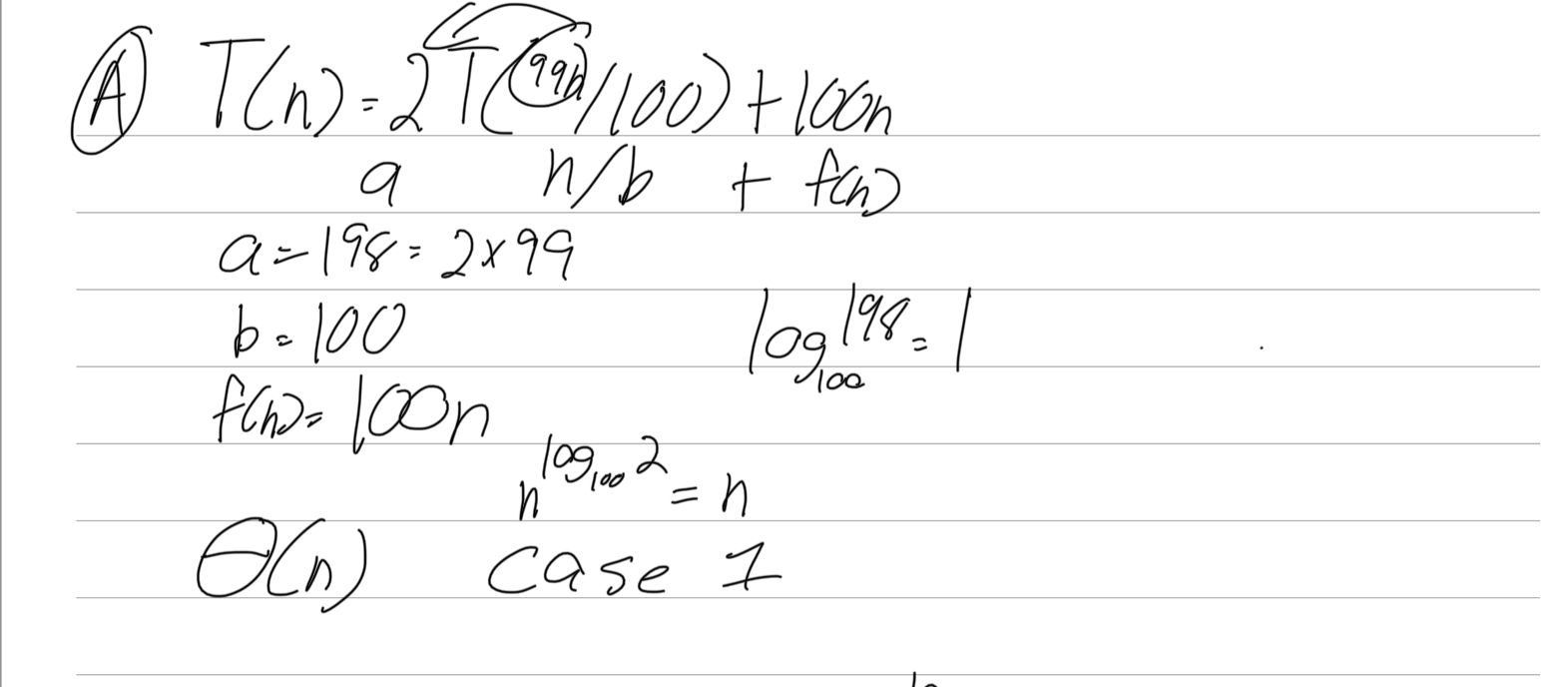
If c = 5 we have T(k+1) < c(k+1)

Value of c:

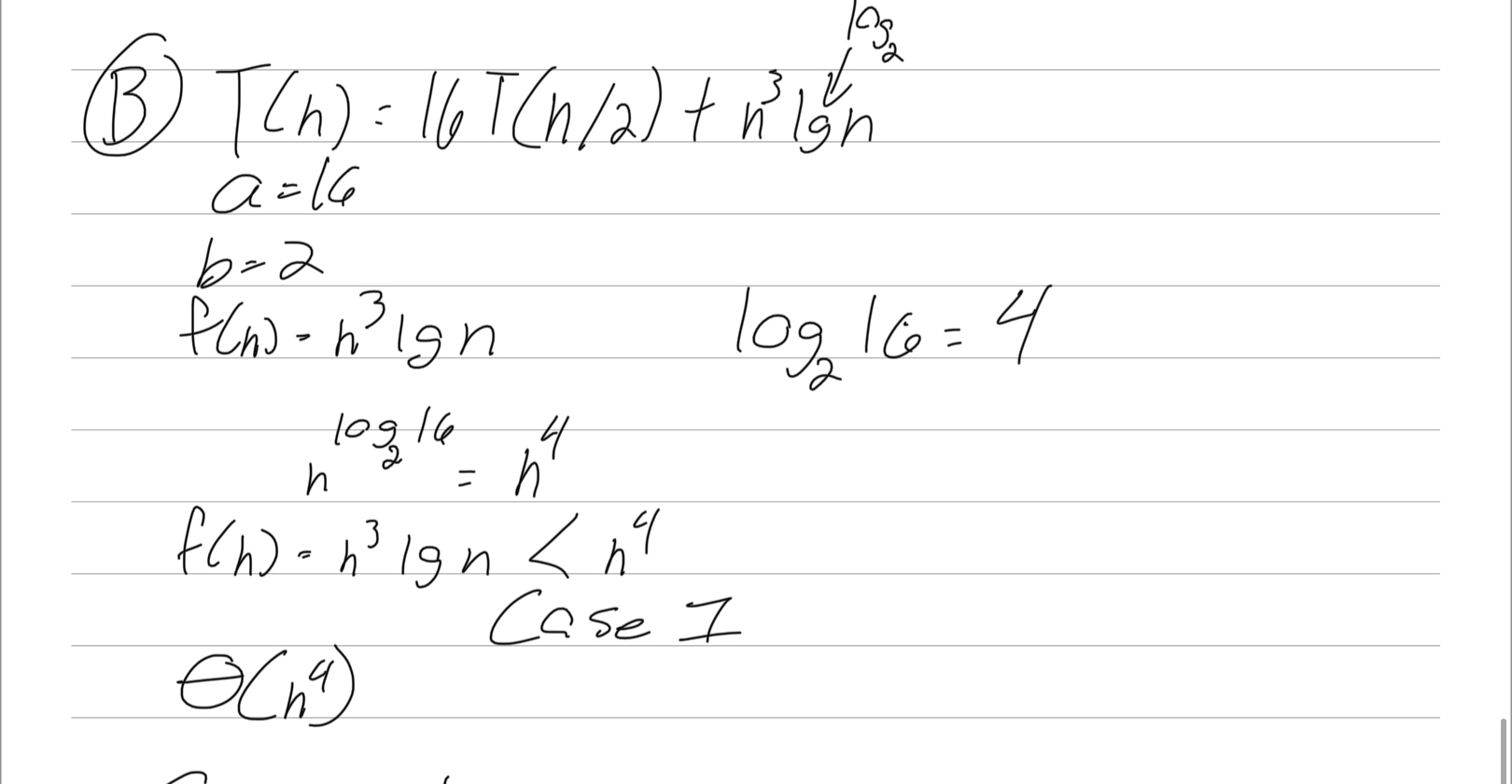
C=5

**7. (15 points)** Use the Master Method to solve the following three recurrence relations and state the complexity orders of the corresponding recursive algorithms.

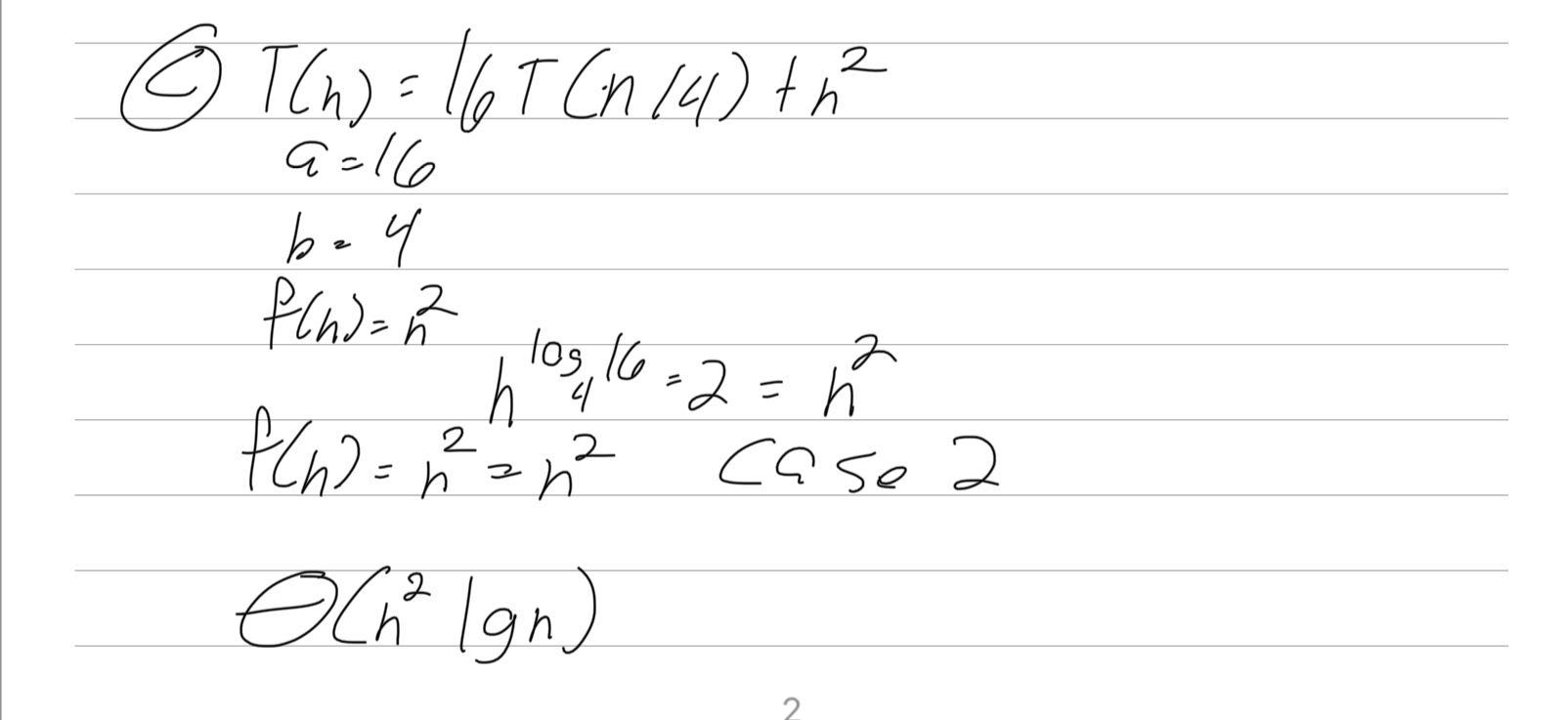
1. T(n)=2T(99n/100)+100n



1. T(n)=16T(n/2)+n3lgn



1. TT(n)=16T(n/4)+n2



**8. (20 points)** Use Backward Substitution (10 points) and then Forward Substitution (10 points) to solve the recurrence relations T(n)=2T(n-1)+1;T(0)=1. In each case, do the following: (1) Show at least three expansions so that the emerging pattern is evident. (2) Then write out T(n) fully and simplify using equation (A.5) on Text p.1147. (3) Verify your solution by substituting it in the LHS and RHS of the recurrence relation and demonstrating that LHS=RHS. (4) Finally, state the complexity order of T(n). You must show your work for parts (1)-(3) to receive credit.

