Problem 1 (10 Points)

The following algorithm is *linear search* , where $a_1, a_2, \ldots, a_n$: are not necessarily distinct
In particular, *location* is the subscript of the first occurrence of a term that equals x or is 0 if x is not found:

```
procedure linear search (x : integer, a1, a2,…, an: integers)
1       i := 1 {i is left endpoint of search interval}
2       while (i ≤ n and x ≠ ai )
3               i := i + 1
4       if i > n then location := 0
5       else
6       begin
7               location := - i
8               i := i + 1
9               while (i ≤ n and x ≠ ai )
10                      i := i + 1
11              if i ≤ n then location := i
12      end
```

a. (3) Complete the pseudo code in line 3 above

b. (7) Adjust in the space above, the algorithm to compute *location* to be the subscript of the **second occurrence** of a term that equals x, or is the **negative** of the subscript of the first and only occurrence of x if x is found only once, or 0 if x is not found.
(e.g. if the list is 3, 2, 1, 2, 8, 6, 2, 10 then for $x = 5$, *location* = 0; for $x = 1$, *location* = -3; for $x = 2$, *location* = 4)
Explain your adjustment in English in the space below

Line 4 will return 0 in *location* if x is not found.
Else, x is found and its first occurrence is at index i. The negative of i is assigned to *location* in line 7.
In lines 8-9-10, the search continues from just after this first occurrence of x (line 8, where i is incremented). If x is found again (line 11), *location* will be updated to the subscript at which x is found the **second time**. Else, location is not updated, and remains to hold the negative of the subscript of where x was found first.

Problem 2 (10 Points)
The following algorithm is *binary search*, where $a_1, a_2, \ldots, a_n$ are in non-decreasing order, and not necessarily distinct; In particular**,** *location* is the subscript of the term that equals x or is 0 if x is not found

procedure *binary search* (x : integer, $a_1, a_2, \ldots, a_n$: non-decreasing integers)
1 $i := 1$ {i is left endpoint of search interval}
2 $j := n$ {j is right endpoint of search interval}
3 **while** $i < j$
4 **begin**
5 $m := \lfloor (i+j)/2 \rfloor$
6 **if** $x > a_m$ **then** $i := m+1$
7 **else** $j := m$
8 **end**
~~9 **if** $x = a_i$ **then** *location* $:= i$~~
~~10 **else** *location* $:= 0$~~
9 **if** $x \neq a_i$ **then** *location* $:= 0$ {x is not found}
10 **else begin**
11 *location* $:= -i$ {first occurrence of x}
12 **if** $i < n$ **then** {there won't be another occurrence of x if $i=n$}
13 **if** $x = a_{i+1}$ **then** *location* $:= i + 1$
14 **end**

a. (3) Complete the pseudo code in line 5 above
b. (7) Adjust in the space above, the algorithm to compute *location* to be the subscript of the **second occurrence** of a term that equals x, or is the **negative** of the subscript of the first and only occurrence of x if x is found only once, or 0 if x is not found.
(e.g. if the list is 1, 2, 2, 2, 6, 8, 10 then for $x = 5$, *location* = 0; for $x = 1$, *location* = -1; for $x = 2$, *location* = 3)
Explain in English in the space below your adjustment of the algorithm

Two main observations:

1. If x is present in more than one location in the given list, these locations have to be adjacent successive positions, since the list is non-decreasing. So if the first occurrence is at position i, then the second occurrence if any must be at $i+1$.
2. At the end of the while loop, if x is found at subscript i then this will be the subscript of the **first** occurrence of x in the list. This is because in line 7, even if $x = a_m$ the search does not stop, but it continues to "zoom in" until $i = j$.

So the adjustment is as follows: If x is not found, the *location* is assigned 0 (Line 9).
Otherwise, if x is found, then the first occurrence is at i. So set *location* to $-i$ (Line 11).
Note that if $i = n$ then there will not be a second occurrence. Otherwise, if $i<n$, then if there is another occurrence it has to be at i+1, since the list is non-decreasing. So all that is needed is to check if $x = a_{i+1}$., and if so, set *location* to i+1. (Lines 12-13)

.

Problem 3 (10 Points)
This problem also refers to the Binary Search algorithm given above.
Write the binary search algorithm as a recursive algorithm.
The idea is:
Basis step: If the array contains one element, then check if that element is x. If it is, the return the location of that element; if not then return 0 in *location.*
Recursive step:
If x > the middle element then Binary Search in the "upper" half of the list
Else Binary Search in the "lower or equal" part of the list.

procedure *binary search recursive* (x integer; , $a_1, a_2, \ldots, a_n$: non-decreasing integers; i, j: subscripts representing the boundaries of where to search next)

if $i=j$ **then** **{base case}**
  if $x = a_i$ **then** *location* := i **else** *location* = 0
else begin
  $m := \lfloor (i+j)/2 \rfloor$
  if $x > a_m$ **then** *binary search recursive* (x ; $a_1, a_2, \ldots, a_n$; $m+1, j$)
  else *binary search recursive* (x; $a_1, a_2, \ldots, a_n$; i, m)
end

Problem 4 (15 Points)

a. (5) Use the definition of big-oh to show that $2n^2 - 100n$ is $O(n^2)$. Make sure to give the witnesses.

For any $n > 1$, $|2n^2 - 100n| \leq 2n^2 + 100n \leq 2n^2 + 100n^2 \leq 102n^2$. So $2n^2 - 100n$ is $O(n^2)$ with witnesses $k = 1$ and C = 102.

b. (5) Use the definition of big-omega to show that $2n^2 - 100n$ is $\Omega(n^2)$. Make sure to give the witnesses

Note that $2n^2 - 100n \geq 0$ for $n \geq 50$. So $|2n^2 - 100n| = 2n^2 - 100n = n^2 + (n^2 - 100n)$
$\geq n^2$ for $n \geq 100$
Thus $2n^2 - 100n$ is $\Omega(n^2)$, where C=1, and k =100 are witnesses.

c. (5) Conclude that $2n^2 - 100n$ is $\Theta(n^2)$.

By the definition of big-theta, since $2n^2 - 100n$ is $O(n^2)$ and $2n^2 - 100n$ is $\Omega(n^2)$, then $2n^2-100n$ is $\Theta(n^2)$.

Problem 5 (15 Points)

Consider the proposition

$P(n)$: An amount of postage of n cents can be formed using 3-cent and 5-cent stamps.

You should prove that $P(n)$ *is true* for $n \geq 8$, first using mathematical induction and then using strong mathematical induction.

a. (8) Use mathematical induction to prove that $P(n)$ *is true* for $n \geq 8$.

Basis step: $P(8)$ is true, since $8 = 3.1 + 5.1$

Inductive Step: Assume $P(k)$ is true. Show that $P(k+1)$ is true.

So $k = 3.a + 5.b$. Must show that $k+1 = 3.a' + 5.b'$

Case 1: $b = 0$

$k = 3.a$. Since $k \geq 8$, then $a \geq 3$. So $k+1 = 3.a + 1 = 3.(a-3) + 9 + 1 = 3.(a-3) + 10 = 3.(a-3) + 5.2$. SO $P(k+1)$ is true.

Case 2: $b > 0$

$k = 3.a + 5.b$. So $k + 1 = 3.a + 5.b + 1 = 3.a + 5. (b-1) + 5 + 1 = 3.(a+2) + 5.(b-1)$. So $P(k+1)$ is true.

Thus by mathematical induction $P(n)$ *is true* for $n \geq 8$.

b. (7) Use strong mathematical induction to prove the result. (Hint: Show that the statements $P(8)$, $P(9)$, and $P(10)$ are true, and use strong induction accordingly)

BASIS STEP: $P(8)$, $P(9)$, and $P(10)$ are true, since $8 = 3.1 + 5.1$, so P(8) is true; $9 = 3.3$ so P(9) is true; $10 = 5.2$ so P(10) is true.

INDUCTIVE STEP: Now assume that $P(8)$, $P(9)$, … $P(k)$, $k \geq 10$ are all true. Show that $P(k+1)$ is true. If $k \geq 10$, then $k-2 \geq 8$. So $P(k-2)$ is true. i.e. $k-2 = 3.a + 5.b$. Now, $k+1 = (k-2) + 3$. But $k-2 = 3.a + 5.b$. So $k+1 = 3.a + 5.b + 3 = 3.(a+1) + 5.b$. Therefore, $P(k+1)$ is true.

Thus by strong mathematical induction $P(n)$ is true for $n \geq 8$.

Problem 6 (10 Points)

In the questions below give a recursive definition with initial condition(s).

a. (4) The function $f(n) = 3^n$, $n = 1, 2, 3, \ldots$

Basis step: $f(1) = 3$

Recursive step: $f(n) = 3.f(n-1)$, $n \geq 2$.

b. (3) The sequence $a_1 = 24, a_2 = 20, a_3 = 16, a_4 = 12, \ldots$

Basis step: $a_1 = 24$

Recursive step: $a_n = a_{n-1} - 4$, $n \geq 2$.

c. (3) The set $\{0, 1, 3, 7, 15, 31\ldots\}$.

Basis step: $0 \in S$

Recursive step: $x \in S \rightarrow 2x+1 \in S$

Problem 7 (10 Points)
Each of the following statements is FALSE. In each case, give a counter example.

a. (4) For all a>1, a^n is $O(2^n)$

Counter example: a=4. Then $a^n = 4^n = 2^n.2^n$ which cannot be bounded by $C2^n$; i.e. we cannot have $2^n.2^n \le C2^n$, since no matter how large C is, 2^n will be larger than C for sufficiently large n… specifically for $n > \log_2 C$.

b. (3) If $f(n)$ is $O(g(n))$ then $g(n)$ is $O(f(n))$

Let $f(n) = n$ and $g(n) = n^2$. Then clearly $f(n)$ is $O(g(n))$ but $g(n)$ is not $O(f(n))$

c. (3) If $f(n)$ is $O(n^2)$ and $g(n)$ is $O(n^2)$, then $f(n) - g(n) = 0$

Let $f(n) = n^2$ and $g(n) = n^2$-2 . Then clearly $f(n)$ is $O(n^2)$ and $g(n)$ is $O(n^2)$, but $f(n) - g(n) = 2$ which is not 0.

Problem 8 (25 Points)
Let $n > 0$ be any integer. Then there is an k such that $2^k \le n < 2^{k+1}$. i.e. n is sandwiched between two successive powers of 2.

a. (5) Complete the following table

n	4	7	18	32	60	90	150
k	2	2	4	5	5	6	7

b. (5) Show that $k = \lfloor log\ n \rfloor$, where log is the logarithmic function base 2.

Let $2^k \le n < 2^{k+1}$. Since the log is an increasing function, it follows that $log\ 2^k \le log\ n < log\ 2^{k+1}$. So $k \le log\ n < k+1$. Thus $k = \lfloor log\ n \rfloor$.

c. (8) Give an algorithm that computes k as above for a given n. Of the arithmetic operations your algorithm can **only** use addition. It cannot use multiplication nor division.

procedure *compute* (n: positive integer)
    $p := 1$
    $k := 0$
    while $p \le n$
        begin
            $p := p + p$
            $k := k + 1$
        end
    $k := k - 1$
{ k is the required power of 2 }

d. (4) What is the operation count of your algorithm of part (c) above? Give the number of additions performed.

$$T(n) = 2\,(\lfloor log\ n \rfloor + 1) + 1$$

e. (3) Based on your answer in part (d), what is the big-O estimate that you can give ?

$$T(n) = O(log\ n)$$