**Problem 1**
(a) $3n^3 + 75n^2 + 8log_2n \epsilon O(n^3)$
(b)1 + 3 + 5 + ... + (2n - 1) $\epsilon O(n^2)$
(c) Show that $1 + 2 + 4 + 8 + ... + 2^n$ is $O(2^n)$
(d) Show that $1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^n} isO(1)$

**Solution**

(a) f(n) = $3n^3 + 75n^2 + 8log_2n \epsilon O(n^3)$

$$75n^2 \leq n^3$$
$$75 \leq n$$

$$8n \leq n^3$$
$$8 \leq n^2$$
$$\sqrt{8} = 2.8 = n$$

$\Rightarrow \quad \leq 3n^3 + n^3 + n^3$
$\Rightarrow \quad 5n^3$
Which also can be seen as a format of c $\cdot n^3$

Let $n_0 = 75, c = 5$ for all n $\geq n_0$

$$3n^3 + 75n^2 + 8log_2n$$
$$\Rightarrow \quad n^2 \cdot n$$
$$\Rightarrow \quad 75 \geq 75$$

$$\Rightarrow \quad n \cdot n^2$$
$$\Rightarrow \quad n^2 \geq n^2$$

$== 5625 \geq 64$

Therefore, it proved that f(n) $\epsilon O(n^3)$

(b) $1 + 3 + 5 + ... + (2n - 1) \epsilon O(n^2)$

We know that the equation = $n^2$
So for O(n) $\Rightarrow 0 \leq T(n) \leq c \cdot g(n)$

$$T(n) = n^2 \leq n^2 \Rightarrow \quad \leq c_1 \cdot n^2$$

*Let* $c_2 = 1, T(n) = 0(n^2)$
So as $\Omega(n^2)$ which $g(n) \leq T(n), but \geq 0$

$$T(n) = n^2 \geq 0$$

$$\geq c_2 \cdot n^2$$

if let $c_1 = 1$
T(n) will equal to $\Omega(n^2)$

For all n$\geq n_0$, let $c_1 = 1$ and $c_2 = 0$
$1 + 3 + 5 + ... + (2n-1) \leq c_2 \cdot n^2 = 1 \cdot n^2 == c_1 \cdot n^2 = 1 \cdot n^2 \leq T(n)$

(c) Show that $1 + 2 + 4 + 8 + ... + 2^n$ is $O(2^n)$

This equation is in the form of geometric series:

$$a + ar + ... + ar^{n-1} = \frac{a[r^n - 1]}{[r-1]}$$

which in this case r = 2 and a = 1, when we plug in it will be as below:

$$2^n - 1 = \frac{1[2^n - 1]}{[2-1]}$$

Now using $lim_{n\to\infty}\frac{f(n)}{g(n)}$, we will get:

$$lim_{n\to\infty}\frac{2^n}{2^n} - \frac{1}{2^n}$$

$\Rightarrow \quad 1 - \frac{1}{2^\infty} == 1 - 0 = 1$

The equation has been proved by showing $1 > 0 \Rightarrow$ 1 + 3 + 5 + ... +(2n-1) is O($2^n$)

(d) Show that $1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^n} is O(1)$

$\sum_{n=1}^{\infty}(x_n)$

Ratio which is also term as r:

$$\frac{x_n + 1}{x_n}(r = \frac{1}{2})$$

$|r| = \frac{\frac{1}{2}}{1} \leq$ is equal to $\frac{a}{1-r}$

$\Rightarrow \frac{\frac{1}{2}}{1-\frac{1}{2}}$

$\Rightarrow 1$

Therefore, the equation 1 + 2 + 4 + 8 + ... + $2^n$ is proved to have running time of O(1)

**Problem 2**
1: for i = 1 to A.length -1
2: for j =A.length down to i+1
3: if A[j] ¡ A[j-1]
4: Exchange A[j] with A[j-1]
(a)Let A' denote the output of $BUBBLE_SORT(A)$. To prove that $BUBBLESSORT$ is correct, we need to first show that it terminated and that

$$A'[1] \leq A'[2] \leq ... \leq A'[n]$$

where n = A.length. In order to show that $BUBBLESORT$ actually sorts, what else do we need to show?
(b) State precisely a loop invariant for the for loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in class.
(c) Using the termination condition of the loop invariant proven in part (b), state another loop invariant for the for loop in lines 1–4 that will allow you to prove inequality (1). Your proof should use the structure of the loop invariant proof presented in class.
(d) What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

**Solution**

(a) In bubble sort each iteration the algorithm checks the element whether the first element is smaller than the later element or not, if not then they are swapped. It is very easy to prove bubble sort is correct by looking at the array of the output which should be sorted and must be element in the original element in A. By that it is already given that element in A' is sorted from A which obviously contain element in A, it's enough to prove that A' contains all element in A

(b) The purpose of line 2-4 is to make the smallest element of sub array to A[i,n] at$i^{th}$ position. In the start of for loop at line 2 the value of j is equal to n. The sub array A will have only one element which is A[n] and smallest compare to j and n. That why the loop invariant holds.In addition line 3 compares the $j^{th} and j-1^{th}$ element of A in each iteration. When it's detected smaller line 4 will swaps them and swapping won't alter the permutation so loop invariant will still hold.

(c) Each iteration of the loop in line 1-4, the sub array contain i-1 elements. At the start of the loop, the value of i = 1. The sub array $A[1..i-1]$ is empty. So the loop invariant holds.
As you can see line 2-4 store the smallest element at i, and by every iteration i is increased by 1 after $i^{th}$ iteration, the sub array A contains i number of smallest element of A in sorted order.

(d) The worst case running time of bubblesort is $O(n^2)$. Which when compare to the insertion sort the worst case for it is also same as $O(n^2)$. However, when comparing the best case bubblesort stays the same where as insertion sort become $O(n)$

**Problem 3**
For some fixed positive integer c, consider the summation

$$S(n) = 1^c + 2^c + 3^c + ... + n^c$$

(a) Show that S(n) is O($n^{c+1}$)
(*b*) Show that S(n) is $\Omega(n^{c+1})$

**Solution**

(a) Show that S(n) is O($n^{c+1}$)

Let show that S(n) is O($n^{c+1}$)
$n_0 = 0, c = 1, n \geq n_0$

As you can see when we plug in the number to the equation we all get:

$$1^c = 1$$
$$2^c = 2$$
$$n^c \cdot n \geq S(n)$$

S(n) $\leq n^{c+1}$, therefore S(n) = O($n^{n+1}$) is True

(b) Show that S(n) is $\Omega(n^{c+1})$

Let c = $\frac{1}{2^{c+1}}$, $n_0 = 0$ for all n $\geq n_0$

$$S(n) \geq (\frac{n}{c})^c + (\frac{n}{c})^c + .... + (\frac{n}{c})^c$$
$$S(n) \geq \frac{n}{2} \cdot (\frac{n}{c})^c \quad \Rightarrow \quad (\frac{n}{c})^{c+1}$$
$$S(n) \geq \frac{1}{2^{c+1}}(n)^{c+1} \quad \Rightarrow \quad c \cdot n^{c+1}$$

**Problem 4**
Recall the definition of logarithm base two: saying $p = log_2 m$ the same as saying $m = 2^p$. In this class, we will typically write log to mean $log_2$.
(a) How many bits are needed to write down a positive integer n?
(b) How many times does the following piece of code print "hello"? Assume n is an integer, and that division rounds down to the nearest integer. Give your answer in big-O notation, as a function of n.

$$while n \geq 1 :$$
$$print "hello"$$
$$n := n/2$$

(c) In the following code, subroutine A(n) takes time $O(n^3)$. What is the overall running time of the loop, in big-O notation as a function of n? Assume that division by two takes linear time.

$$while n \geq 1 :$$
$$A(n)$$
$$n := n/2$$

**Solution**

(a) $\log_2(n)$
(*b*) It will print log(n) times. For the running time it is O(log(n)).
(*c*) Overall running time = O(log(n $\cdot n^3$))

**Problem 5**
Suppose we are given an array A[1..n] of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < ... < A[n]$.
(a) Describe a O(log n)algorithm that either computes an index i such that A[i] = i or correctly reports that no such index exists. Also provide a brief justification of your algorithm.
(b) Suppose we know in advance that A[1] > 0. Describe an even faster algorithm that either computes an index i such that A[i] = i correctly reports that no such index exists.

**Solution**

(a) Let make up Array B as

$$B[i] = A[i] - i$$

we know that $A[i] < A[i+1]$
so we get:

$$B[i] = A[i] - i \leq A[i+1] - 1 - i = B[i+1]$$

which the array B is sorted and not decreased.
Index at B[i] = 0 is equivalent to index at A[i] = i, which is enough for us to binary search for 0 in B. Moreover, the running time is O(log n).

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the
        // middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not present
    // in array
    return -1;
}

Credit: https://www.geeksforgeeks.org/binary-search/
```

**Problem 6**
(a) Describe and analyze an algorithm to compute the unknown integer k. Your algorithm should be strictly better O(n)
(b) Describe and analyze an algorithm to determine if the given array contains a given number x. Again, your algorithm should be strictly better O(n)

**Solution**

(a) The binary search would come in play when we are dealing with this kind of problem. At first finding k is just to scan the whole array as the number to the left is lesser compare to the number on the right. From the above method running time would be simply O(n). However, for a better running time we could divide the array into 2 part and one will be sorted and one will not be sorted. As we all know k will be in the sorted list so that why we will now use the binary search to get the better running time of O(log(n))

(b) After we have found K, to find x we just have to use root-finding method or known as bisection which the running time is O(log(n))