aryanrao-HW1

1.

(a) $8n^3 + 9n^2 + 5 \in O(n^3)$

Proof:

$f(n) = 8n^3 + 9n^2 + 5 \leq 8n^3 + 9n^3 + 5n^3$

$f(n) = 8n^3 + 9n^2 + 5 \leq 22n^3$

$= c \cdot n^3$

$= c \cdot g(n)$

Choose c = 22, N = 1

(b) $2^{2^{n+2}} \in O(2^{2^{n+1}})$

We need to show that for every constant $c \geq 1$ and every threshold $N > 0$, there exists an $n > N$ for which $f(n) \geq c.g(n)$. Alternatively, we can use a proof by contradiction.

Assume there exist constants $c \geq 1$ and $N > 0$ such that for all $n > N$, $f(n) < cg(n)$.

Let's consider $n = N + 1$. We have:

$f(N + 1) = 2^{2^{N+1+2}} = (2^{2^{N+3}})$

$> (2^{2^{N+2}})$ (since $2^{N+3} > 2^{N+2}$)

Now, let's consider c = 1. For n = N + 1, we have:

$c.g(n) = 1 * g(n) = g(n) = 2^{2^{n+2}}$

Since $f(N + 1) > cg(n)$ (which is $2^{2^{N+2}}$)this contradicts our assumption.

Therefore, there does not exist any constant $c \geq 1$ and threshold $N > 0$ for which $f(n) < cg(n)$ holds for all $n > N$. This implies that $f(n) = 2^{2^{n+2}}$ does not belong to $O(2^{2^{n+1}})$

(c) To prove that if $f \in O(g)$ and h is any positive-valued function, then $fh \in O(gh)$:

Since $f \in O(g)$, there exists a constant $c \geq 1$ such that for every $n \geq 1$, $f(n) \leq cg(n)$.

Now, we can prove that $fh \in O(gh)$ by showing that for the same constant c, there exists a constant $c_2 \geq 1$ such that for every $n \geq 1$, $fh(n) \leq c_2gh(n)$.

Let's consider $c_2 = c$. For every $n \geq 1$, we have:

$fh(n) = f(n)h(n) \leq cg(n)h(n)$

(since $f(n) \leq cg(n)$ for every $n \geq 1$)

$= c(gh(n))$

Therefore, for $c_2 = c$ and $n \geq 1$, we have $fh(n) \leq c_2gh(n)$.

This proves that $fh \in O(gh)$.

2.

(a) r = 0;

```
for i in the range [1, n] {
    for j in the range [i, n]
        for k in the range [1, j-i]
            r++;
print (r);
}
```

Suppose the 'k' loop takes c1 time per iteration, 'j' loop takes c2 and 'i' loop takes c3 respectively.

T(n) =

$$\sum_{i=1}^{n}(c3 + \sum_{j=i}^{n}(c2 + \sum_{k=1}^{j-i} c1))$$

Expanding,

T(n) =

$$\sum_{i=1}^{n} c3 + \sum_{i=1}^{n}\sum_{j=i}^{n} c2 + \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=1}^{j-i} c1$$

T(n) = $c3.n \; + \frac{1}{2} n(n+2).\, c2 + \frac{1}{6}n(n^2 -1).\, c1$

T(n) = $c3.n \; + c2(\frac{n^2+n}{2}) + c1(\frac{n^3-n}{6})$

T(n) = $c3.n + c2(\frac{n^2}{2}) + c2(\frac{n}{2}) + c1(\frac{n^3}{6}) - c1(\frac{n}{6})$

Because n^3 is of the highest order, T(n) = $O(n^3)$

(b) x = Math.pow(2, n)

```
for (i = 1; i <=x; i = i*2)
    for j in range [1, i]
        Constant Number of Operations
```

Outer loop (i) = 1to x, $i = i * 2$
Inner loop (j) = 1 *to i*
x = 2^n, $n = \log_2 x$
x = $\log_2 2^n$ = n

T(n) of outer loop = O(n)
Inner loop: number of iterations = 1 to $i = O(i)$

$i = i * 2$:
$2^0 + 2^1 + 2^2 + \ldots + 2^{n-1}$

The number of iterations is 2^{n-1}. Hence, $T(n) = O(2^n)$
The total efficiency: $O(n + 2^n) = O(2^n)$

(c) i = n
while i >= 1
for j in range [1, i]
Constant Number of Operations
i = i / 2

Outer loop will run as long as 'i' is greater than or equal to 1.
Inner loop is from 1 to 'i.'
While loop: $i = \frac{n}{2}$ halves each time.

The outer loop is having a complexity: $O(\log n)$
The inner loop is having a complexity: i = n , $\frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}$ ....

The number of iterations in each iteration of the outer loop decreases as i gets divided by 2. In the first iteration, the inner loop runs n times, then n/2 times, then n/4 times, and so on. The total number of iterations for the inner loop can be approximated as the sum of the geometric series n + n/2 + n/4 + ... +2n, which is 2n. Therefore, the runtime of the inner loop is O(n).

Combining these steps, the overall runtime of the given algorithm as a function of n is O(log n) * O(n) = O(n log n)

Therefore, the algorithm's runtime has a Big-O upper bound of O(n log n).

3.

```
function polynomial( A, t){ //function to calculate the polynomial
n = length[A]
p_total = 0 // variable to return the total value of polynomial
for i = 0 to n-1{ //assuming stating index is 0
  p_total = p_total + A[i] * Math.pow( t , i)// Math.pow is not a primitive function
}
return p_total //return final value
}
```

The Big- O for this would be O(n) because the for loop will run 'n' times.
Big-O = O(n)

4.

```
function findMedian(A) {
   int n = A.length
   sort(A); // Sort the array in non-decreasing order

   if n % 2 = 1{
      // Array size is odd
      return A[n / 2]; // Return the middle element as the median
   }
else {
      // Array size is even
      int mid1 = n / 2
      int mid2 = mid1 - 1
      return (A[mid1] + A[mid2]) / 2 // Return the average of the two middle elements
as the median
   }
}
```

Pseudo code for a sort method as runtime is dominated by it:

```
Function MergeSort(A):
   if length(A) <= 1:
      return A

   mid = length(A) / 2
   left = A[0:mid]      // Divide the array into two halves
   right = A[mid:length(A)]

   // Recursively sort the two halves
   left = MergeSort(left)
   right = MergeSort(right)

   return Merge(left, right)    // Merge the sorted halves

Merge(left, right):
   merged = []     // Create an empty array to store the merged result
   i = 0           // Index for the left array
   j = 0           // Index for the right array

   // Compare elements from the left and right arrays and merge them in sorted order
   while i < length(left) and j < length(right):
      if left[i] <= right[j]:
         merged.append(left[i])
         i = i + 1
      else:
         merged.append(right[j])
         j = j + 1

   // Append any remaining elements from the left array
```

```
    while i < length(left):
        merged.append(left[i])
        i = i + 1

    // Append any remaining elements from the right array
    while j < length(right):
        merged.append(right[j])
        j = j + 1

    return merged
```

Sorting the array A takes O(n log n) time in the worst case, where n is the size of the array.
Returning the middle element or the average of two middle elements takes constant time, denoted as O(1).
Therefore, the overall runtime of the algorithm is O(n log n) due to the sorting step, which dominates the runtime.
Hence, the worst-case runtime of the algorithm, expressed as a function of the size of the input array, is O(n log n).