Problem 1: Assume that \( k, \epsilon \) are constants with \( k \geq 1 \) and \( 0 < \epsilon < 1 \). Also \( \log \) denotes the logarithm in base 2, and \( \ln \) the natural logarithm in base \( e \). State which among \( A = O(B) \), \( A = \Omega(B) \) and \( A = \Theta(B) \) is correct for each pair of function below. Justify your answer. 2 points per question.

- \( A = \log^k n \) and \( B = n^\epsilon \)
- \( A = n^k \) and \( B = (n^{\log n})^{\ln n} \)
- \( A = 2^n \) and \( B = 3^n \)
- \( A = n^{\log n} \) and \( B = 2^{\log^3 n} \)
- \( A = 3^n \) and \( B = 2^{n^2} \)

Solution:

- \( \log^k n = O(n^\epsilon) \). Actually we have that \( \log^k n = o(n^\epsilon) \) since \( \lim_{n \to \infty} \log^k n / n^\epsilon = 0 \) for any \( k \geq 1 \) and \( 0 < \epsilon < 1 \). Note that this means that any power of the log function (no matter how big) still grows slower than any \( \delta \)-root of \( n \) (no matter how big \( \delta = \epsilon^{-1} \)).
- \( n^k = O((n^{\log n})^{\ln n}) \). Actually we have that \( n^k = o((n^{\log n})^{\ln n}) \) since \( k \) is a constant and \( \log n \ln n \) is a monotonically increasing function of \( n \).
- \( 2^n = O(3^n) \). Actually \( 2^n = o(3^n) \) since \( \lim_{n \to \infty} 2^n / 3^n = \lim_{n \to \infty} (3/2)^n = 0 \).
- Remember that \( 2^{\log n} = n \), therefore \( 2^{\log^2 n} = n^{\log^2 n} \) and therefore \( B = A^{\log n} \). So we have that \( A = o(B) \) (and therefore also \( A = O(B) \)) since \( \lim_{n \to \infty} A / B = \lim_{n \to \infty} A^{1 - \log n} = 0 \).
- Note that \( 3^n = 2^{cn} \) where \( c = \log 3 \) which is a constant. So we are comparing \( 2^{cn} \) to \( 2^{n^2} \), since \( cn = o(n^2) \) we have that \( 3^n = o(2^{n^2}) \) and therefore \( 3^n = O(2^{n^2}) \).

Problem 2: For each of the following recurrences: (i) describe what kind of ”divide and conquer” algorithm would give rise to such a recurrence; (ii) give asymptotic upper and lower bounds on \( T(n) \). Make your bounds as tight as possible and justify your answer. Assume \( T(n) \) is a constant for \( n \leq 2 \).

- \( T(n) = 3T(n/3) + n \log n \) [3 points]
- \( T(n) = \sqrt{n}T(\sqrt{n}) + n \) [3 points]
- \( T(n) = 2T(n/2) + \frac{n}{\log n} \) [4 points]

Solution:

- This is a divide and conquer algorithm that splits the input into 3 parts of equal size \( n/3 \) and recurs on all of them. The splitting and the recombining of the solution requires \( n \log n \) steps. We cannot apply the Master Method directly to this recurrence since \( n^{\log_3 3} = n \) but the free term of the recurrence is \( n \log n \) which is not \( \Omega(n^{1+\epsilon}) \). So we solve this directly. If we draw a recursion tree we have that
  - at level 0, the root, we pay \( n \log n \).
  - at level 1, we pay \( 3 \cdot (n/3) \cdot \log(n/3) = n \log n - n \)
  - in general at level \( i \) we pay \( 3^i \cdot (n/3^i) \cdot \log(n/3^i) = n \log n - ni \)
Note that we have $\log_3 n$ levels. So the total cost is

$$\sum_{i=0}^{\log_3 n} (n \log n - ni) = n \log n \log_3 n - n \sum_{i=0}^{\log_3 n} i = n \log n \log_3 n - n \frac{\log_3 n (\log_3 n + 1)}{2}$$

Recall that $\log_3 n = a \log n$ where $a = \log_3 2 < 1$. Which implies

$$T(n) = an \log^2 n - \frac{a^2}{2} n \log^2 n - \frac{a}{2} n \log n$$

setting $b = a - \frac{a^2}{2} > 0$ we have

$$T(n) = bn \log^2 n - \frac{a}{2} n \log n$$

which is $\Theta(n \log^2 n)$

- This is a divide and conquer algorithm that splits the input into $\sqrt{n}$ parts of equal size $\sqrt{n}$ and recurs on all of them. The splitting and the recombining of the solution requires $n$ steps. To solve, substitute $n = 2^m$. Then we get

$$T(2^m) = 2^{m/2} T(2^{m/2}) + 2^m$$

and if we set $S(m) = T(2^m)$ we have

$$S(m) = 2^{m/2} S(m/2) + 2^m$$

If we build a recursion tree for this recurrence we have a tree of depth $\log m$ where each node at level $j$ contains $2^{m/2^j}$ input values (starting with $j = 0$ at the root). Therefore at level $j$ we must have $a$ nodes such that $a2^{m/2^j} = 2^m$, i.e.

$$a = 2^m - \frac{m}{2^j} = 2 \frac{2^j - 1}{2^j}$$

Each node does $2^{m/2^j}$ work. So each level does

$$a2^{m/2^j} = 2 \frac{2^j - 1}{2^j} + \frac{m}{2^j} = 2^m$$

work. Since there are $\log m$ levels, the total work is $T(n) = 2^m \log m$. Remember now that $2^m = n$ and therefore $m = \log n$, yielding $T(n) = \Theta(n \log \log n)$.

- This is a divide and conquer algorithm that splits the input into 2 parts of equal size $n/2$ and recurs on all of them. The splitting and the recombining of the solution requires $\frac{n}{\log n}$ steps. Again we cannot apply the Master Method directly to this recurrence since $n \log 2 = n$ but the free term of the recurrence is $\frac{n}{\log n}$ which is not $O(n^{1-\epsilon})$. So we solve this directly. If we draw a recursion tree we have that

  - at level 0, the root, we pay $\frac{n}{\log n}$.
  - at level 1, we pay $2 \frac{n/2}{\log(n/2)} = \frac{n}{\log n - 1}$
  - in general at level $i$ we pay $2^i \frac{n/2^i}{\log(n/2^i)} = \frac{n}{\log n - i}$

Note that we have $\log n - 1$ levels since we must stop at $n = 2$ (the recurrence is not defined for $n = 1$ since $\log 1 = 0$ and we cannot divide for 0). So the total cost is

$$\sum_{i=0}^{\log n - 1} \frac{n}{\log n - i} = n \sum_{i=1}^{\log n} \frac{1}{i}$$

which is $\Theta(n \log \log n)$ since $\sum_{i=1}^{k} \frac{1}{i} = \Theta(\log k)$.  

---
**Problem 3:** Given a set $A$ of $n$ integers and another integer $t$, describe an algorithm that determines whether or not there exists two elements in $A$ such that their sum is exactly $t$. Prove that your algorithm is correct and analyze its running time. Full credit will be given to the fastest algorithm.

**Solution:** One trivial solution is to try all possible pairs of elements of $A$ and see if their sum equals $t$. This requires $O(n^2)$ additions and comparisons. A faster algorithm would be to sort the set $A$ and then search for the pair by comparing $t$ with the sum of the minimum and maximum element of $A$, and discarding either the minimum or the maximum depending on the result. More specifically consider the following algorithm

```
Find-Sum(A,t)
    i ← 1; j ← n;
    B ← Merge-Sort(A);
    While j − i > 0 Do
        If B[i] + B[j] = t Then Return True and Stop;
        If B[i] + B[j] < t Then i ← i + 1;
        If B[i] + B[j] > t Then j ← j − 1;
    End While
    Return False
```

Let’s prove first that this algorithm is correct. Consider the three possible choices inside the While loop. If $B[i] + B[j] = t$ then the algorithm correctly returns True. If $B[i] + B[j] < t$, then since the vector $B$ is sorted then for any $k$ such that $i < k < j$ we have that $B[i] + B[k] < B[i] + B[j] < t$ so we can safely discard $B[i]$ since it will never add up to $t$ with any of the elements left in the array $B$. Similarly if $B[i] + B[j] > t$, then for any $k$ such that $i < k < j$ we have that $B[k] + B[j] > B[i] + B[j] > t$ so we can safely discard $B[j]$ since it will never add up to $t$ with any of the elements left in the array $B$.

To analyze the running time, note that the While loop is executed at most $n$ times since at each executions either the algorithm stops or the difference $j − i$ decreases by 1. The work inside the While loop is constant, so the total cost of the While loop is $O(n)$. Therefore the running time of this algorithm is $\Theta(n \log n)$ since the sorting step with Merge-Sort takes $\Theta(n \log n)$, which dominates the $O(n)$ cost of the While loop.