1

ALGORITHM *Secret*$(A[0..n-1])$

```
//Input: An array A[0..n − 1] of n real numbers
minval ← A[0]; maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] < minval
        minval ← A[i]
    if A[i] > maxval
        maxval ← A[i]
return maxval − minval
```

a. What does this algorithm compute?

b. What is its basic operation?

C How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

e. Is there any better algorithm to this?

Solution to the Given Algorithm Analysis Questions

a. What does this algorithm compute?

This algorithm finds the **range** (difference between the maximum and minimum values) of an array of n real numbers.

- It initializes minval and maxval to the first element of the array.
- It iterates through the array and updates minval whenever it finds a smaller value and maxval whenever it finds a larger value.
- Finally, it returns maxval - minval, which represents the range of the dataset.

✅ **Output:** max(A) - min(A)

---

b. What is its basic operation?

The basic operation of this algorithm is the **comparison** operation:

1. if A[i] < minval (comparison)
2. if A[i] > maxval (comparison)

These are the key operations that determine the time complexity of the algorithm.

✅ **Basic Operation: Comparisons (< and >)**

---

c. How many times is the basic operation executed?

- The algorithm iterates over n-1 elements (from index 1 to n-1).
- In each iteration:
    - It performs **two comparisons** (one for minval, one for maxval).
- Total number of comparisons: 2(n−1)2(n-1)

✅ **Total comparisons executed: 2(n-1)**

---

d. What is the efficiency class of this algorithm?

- The loop runs **O(n)** times.
- In each iteration, it performs **constant (2) operations**.
- Thus, the time complexity is **O(n)**.

✅ **Efficiency Class: O(n) (Linear Time Complexity)**

---

e. Is there a better algorithm for this?

- The given algorithm performs **2(n-1) comparisons**.
- A **better algorithm** is the **"Pairwise Comparison Algorithm"**, which finds both the min and max in **1.5n comparisons** (instead of **2n**).
- **How?**
    - Instead of checking every element twice, process elements in **pairs**:
        1. Compare two elements to determine the local min and max.
        2. Compare the local min with the global min and local max with the global max.
    - This reduces the number of comparisons to approximately **1.5n** instead of **2n**.

✅ **Better Algorithm: Pairwise Comparison (Reduces comparisons to ~1.5n)**

Final Answers

Question	Answer
(a)	Computes the **range** (max - min) of an array.
(b)	Basic operations are **comparisons (<, >)**.

Question Answer

(c) 2(n-1) comparisons are executed.

(d) Efficiency class is **O(n)**.

(e) A better algorithm is **Pairwise Comparison (1.5n comparisons instead of 2n)**.

🚀 Let me know if you need further clarification or a code implementation of the optimized approach!

2

Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n) = 1^3 + 2^3 + \cdots + n^3$.

Algorithm $S(n)$
//Input: A positive integer n
//Output: The sum of the first n cubes
if $n = 1$ **return** 1
else return $S(n\text{-}1) + n * n * n$

a. Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

b. How does this algorithm compare with the straightforward nonrecursive algorithm for computing this function?

Solution for the Given Recursive Algorithm

The given algorithm calculates the sum of cubes of the first `n` natural numbers:

$$S(n) = 1^3 + 2^3 + \ldots + n^3$$

(a) Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Step 1: Define the Recurrence Relation

- The algorithm follows:
  - **Base case**: When `n = 1`, return `1`. (Single operation)
  - **Recursive case**: $S(n) = S(n-1) + n^3$
- The **basic operation** is computing `n^3` and the recursive call.
- Let $T(n)$ be the number of times the basic operation (multiplication and addition) is executed.

Since each call executes `n^3` once and makes a recursive call to `S(n-1)`, the recurrence relation is:

$$T(n) = T(n-1) + O(1)$$

Since each call executes `n^3` once and makes a recursive call to `S(n-1)`, the recurrence relation is:

$$T(n) = T(n-1) + O(1)$$

Step 2: Expand the Recurrence

Expanding the recurrence:

$$T(n) = T(n-1) + 1$$

$$T(n-1) = T(n-2) + 1$$

$$T(n-2) = T(n-3) + 1$$

Expanding until `n = 1`:

$$T(n) = T(1) + (n-1)$$

Since $T(1) = O(1)$, we get:

$$T(n) = O(n)$$

✅ **Final Answer:** The recurrence relation is:

$$T(n) = T(n-1) + O(1)$$

Solving it gives **T(n) = O(n)**.

(b) Comparison with a Non-Recursive Algorithm

A straightforward **non-recursive** way to compute the sum of cubes would be:

python Copy Edit

```python
def sum_of_cubes(n):
    sum = 0
    for i in range(1, n+1):
        sum += i**3
    return sum
```

Comparison

Algorithm Type	Time Complexity	Space Complexity
Recursive	**O(n)** (due to recursive calls)	**O(n)** (stack space for recursion)
Iterative (Loop-based)	**O(n)**	**O(1)**

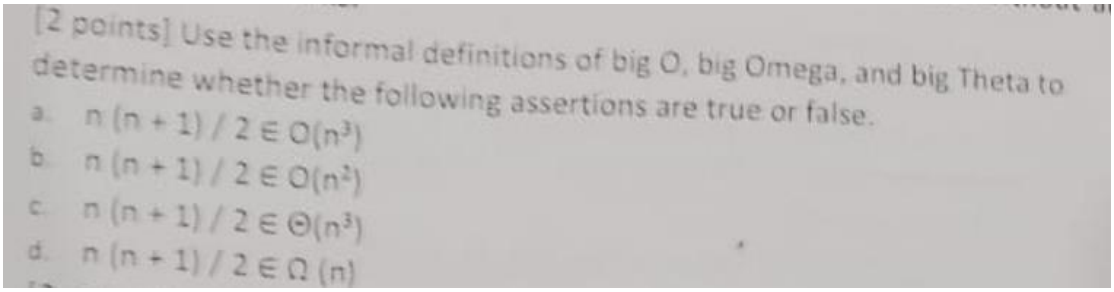
Which One is Better?

- The **iterative version** is **better** because:
  - It avoids the extra memory overhead of recursion.
  - It does not require function calls, making it **faster** in practice.

✅ **Conclusion:** The **non-recursive approach is more efficient** because it **avoids recursion overhead** while maintaining the same time complexity.

Would you like a more optimized mathematical approach to solve this in **O(1)** time? 🚀

3



Below we analyze the function

$$f(n) = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

against each of the common set-membership claims in the question.

---

1. $f(n) \in O(n^3)$

- **Meaning:** "Is $\frac{n^2+n}{2}$ eventually bounded above by a constant times n^3?"
- For large n, $n^2 + n \leq n^2 + n^2 = 2n^2$, and clearly $2n^2 \leq C\, n^3$ for some constant C (e.g., $C = 2$) whenever $n \geq 1$.
- **Verdict: True.** It's trivially upper-bounded by n^3.

---

2. $f(n) \in O(n^2)$

- **Meaning:** "Is $\frac{n^2+n}{2}$ eventually bounded above by a constant times n^2?"
- For large n, $n^2 + n \leq n^2 + n^2 = 2n^2$, so

$$\frac{n^2+n}{2} \leq \frac{2n^2}{2} = n^2.$$

- We can take $C = 1$ and $n_0 = 1$.
- **Verdict: True.** It grows on the order of n^2.

---

3. $f(n) \in \Theta(n^3)$

- **Meaning:** "Is $\frac{n^2+n}{2}$ both $O(n^3)$ and $\Omega(n^3)$?"
- Although it is $O(n^3)$ (from part a), to be $\Theta(n^3)$ it must also be **at least** a constant times n^3 for large n. Formally, we would need

$$\frac{n^2+n}{2} \geq c n^3$$

  for some $c > 0$ and all large n. This is **not** true, because n^2 grows more slowly than n^3.
- **Verdict: False.** It is not in $\Theta(n^3)$.

---

4. $f(n) \in \Omega(n)$

- **Meaning:** "Is $\frac{n^2+n}{2}$ eventually bounded below by a constant times n?"
- For large n, observe

4. $f(n) \in \Omega(n)$

- **Meaning:** "Is $\frac{n^2+n}{2}$ eventually bounded below by a constant times n?"
- For large n, observe

$$\frac{n^2+n}{2} > \frac{n^2}{2}$$

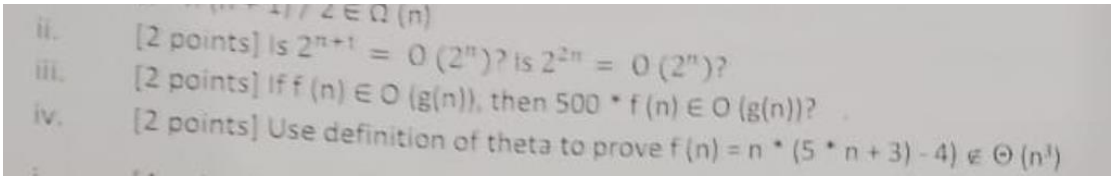
  and $\frac{n^2}{2} \geq cn$ is easy to satisfy if n is large (take $c = n/2$ for instance, or fix $c = 1$ for $n \geq 2$).
- **Verdict: True.** It is at least on the order of n (in fact, it's $\Omega(n^2)$, which is even stronger).

Answer Summary

1. $f(n) \in O(n^3)$: True
2. $f(n) \in O(n^2)$: True
3. $f(n) \in \Theta(n^3)$: False
4. $f(n) \in \Omega(n)$: True

‹ 2/2 ›

4



2. (a) Is $2^{n+1} \in O(2^n)$?

- By definition, $2^{n+1} = 2 \cdot 2^n$.
- To show $2^{n+1} \in O(2^n)$, we need constants $c > 0$ and n_0 such that

$$2^{n+1} \leq c\,2^n \quad \text{for all } n \geq n_0.$$

- Clearly, $2^{n+1} = 2 \cdot 2^n \leq 2 \cdot 2^n$. Choose $c = 2$. This holds for all $n \geq 0$.
- **Answer:** Yes, $2^{n+1} \in O(2^n)$.

(b) Is $2^n \in O(2^n)$?

- Trivially yes, by choosing $c = 1$.
- We have $2^n \leq 1 \cdot 2^n$ for all n.
- **Answer:** Yes, $2^n \in O(2^n)$.

3. If $f(n) \in O(g(n))$, show $500\,f(n) \in O(g(n))$.

- By definition, $f(n) \in O(g(n))$ means there exist constants $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c\,g(n) \quad \text{for all } n \geq n_0.$$

- Multiply both sides by 500:

$$500\,f(n) \leq 500\,c \cdot g(n) \quad \text{for all } n \geq n_0.$$

- Thus $500\,f(n)$ is also bounded above by a constant times $g(n)$.
- **Conclusion:** $500\,f(n) \in O(g(n))$.

4. Prove $f(n) = n\,(5n + 3) - 4 \in \Theta(n^2)$ Using the Definition

First, expand $f(n)$:

$$f(n) \;=\; n(5n+3) - 4 \;=\; 5n^2 + 3n - 4.$$

Step A: Show $f(n) \in O(n^2)$

We need to find constants $c_1 > 0$ and $n_1 > 0$ such that for all $n \geq n_1$:

$$f(n) \;=\; 5n^2 + 3n - 4 \;\leq\; c_1\, n^2.$$

For $n \geq 1$:

$$5n^2 + 3n - 4 \;\leq\; 5n^2 + 3n^2 = 8n^2 \quad (\text{because } 3n \leq 3n^2 \text{ for } n \geq 1).$$

So pick $c_1 = 8$ and $n_1 = 1$. Then for all $n \geq 1$:

$$f(n) \;=\; 5n^2 + 3n - 4 \;\leq\; 8n^2.$$

Thus, $f(n) \in O(n^2)$.

Step B: Show $f(n) \in \Omega(n^2)$

We also need to find constants $c_2 > 0$ and $n_2 > 0$ such that for all $n \geq n_2$:

$$5n^2 + 3n - 4 \;\geq\; c_2\, n^2.$$

For large n, the $5n^2$ term dominates. In fact, for $n \geq 1$,

$$5n^2 + 3n - 4 \;\geq\; 5n^2 - 4n^2 \;=\; n^2 \quad (\text{since } 3n - 4 \geq -4n^2 \text{ for } n \geq 1).$$

Hence choose $c_2 = 1$ and $n_2 = 1$. Then

$$5n^2 + 3n - 4 \;\geq\; n^2 \quad \text{for all } n \geq 1.$$

Thus, $f(n) \in \Omega(n^2)$.

Thus, $f(n) \in \Omega(n^2)$.

Combine Both (O and Ω)

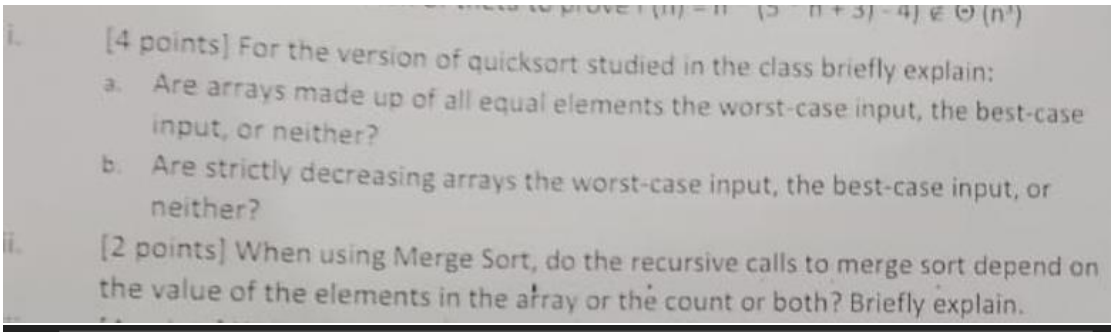
Since $f(n) \in O(n^2)$ and $f(n) \in \Omega(n^2)$, it follows that

$$f(n) \in \Theta(n^2).$$

Answer Summary

1. $\frac{n(n+1)}{2} \in \Omega(n)$
   - **True** (it grows at least as fast as n for large n).
2. $2^{n+1} \in O(2^n)$ and $2^n \in O(2^n)$
   - Both **True** (they differ only by a constant factor).
3. If $f(n) \in O(g(n))$, then $500\, f(n) \in O(g(n))$?
   - **Yes.** Multiplying by a constant does not change the asymptotic class.
4. $f(n) = n(5n+3) - 4 \in \Theta(n^2)$
   - **Yes.** By definition of Θ, we showed it is both $O(n^2)$ and $\Omega(n^2)$.

5(2on paper)



i. [4 points] For the version of quicksort studied in the class briefly explain:
   a. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?
   b. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

ii. [2 points] When using Merge Sort, do the recursive calls to merge sort depend on the value of the elements in the array or the count or both? Briefly explain.

Question i. [4 points] For the version of QuickSort studied in class, briefly explain:

(a) Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

(b) Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

Answer i(a)

For the typical **basic (naïve) QuickSort** version—where the pivot might be chosen as the first or last element—an array of **all equal elements** is a **worst-case** scenario.

- **Why?**
  All elements are equal to the pivot, so one partition ends up with $n - 1$ elements, and the other partition has 0.
  This yields highly unbalanced partitions, leading to $\mathbf{O}(n^2)$ performance.

Answer i(b)

Similarly, a **strictly decreasing** array (reverse-sorted) is also a **worst-case** input for the basic QuickSort approach.

Answer i(b)

Similarly, a **strictly decreasing** array (reverse-sorted) is also a **worst-case** input for the basic QuickSort approach.

- **Why?**
  - If the pivot is the first element, it will be the largest, so everything else goes into the "less-than-pivot" partition ($n - 1$ elements).
  - If the pivot is the last element, it will be the smallest, so everything else goes into the "greater-than-pivot" partition ($n - 1$ elements).
    Either way, it again yields highly unbalanced splits and $\mathbf{O}(n^2)$ time.

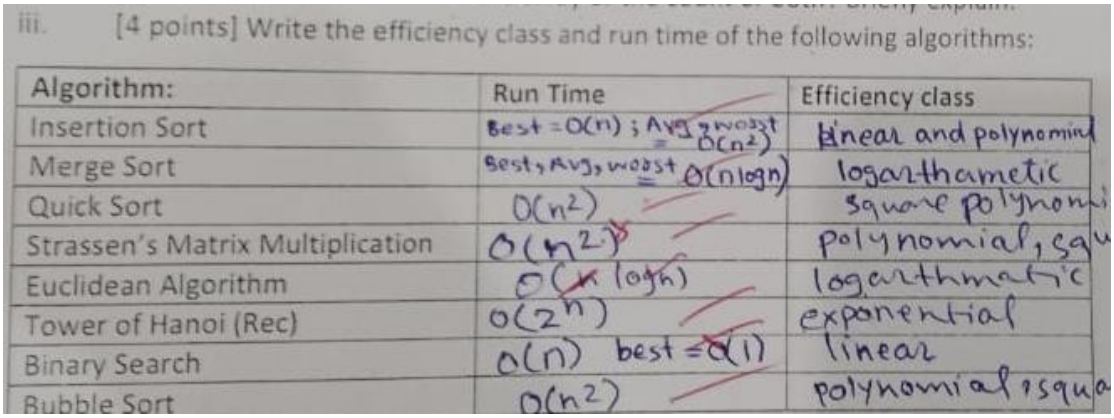
Question ii. [2 points] When using Merge Sort, do the recursive calls to merge sort depend on the value of the elements in the array or the count (size) or both? Briefly explain.

Answer ii

In **Merge Sort**, the division into subproblems depends **only on the size** (count of elements), **not** on the element values.

- **Why?**
  Merge Sort is a **divide-and-conquer** algorithm that always splits the array roughly in half—regardless of the actual values. Each half then recurses independently. This means the structure of the recursion tree depends only on how many elements are in each subarray, **never on how those elements are arranged** (whether sorted, reversed, or random).

6



iii. [4 points] Write the efficiency class and run time of the following algorithms:

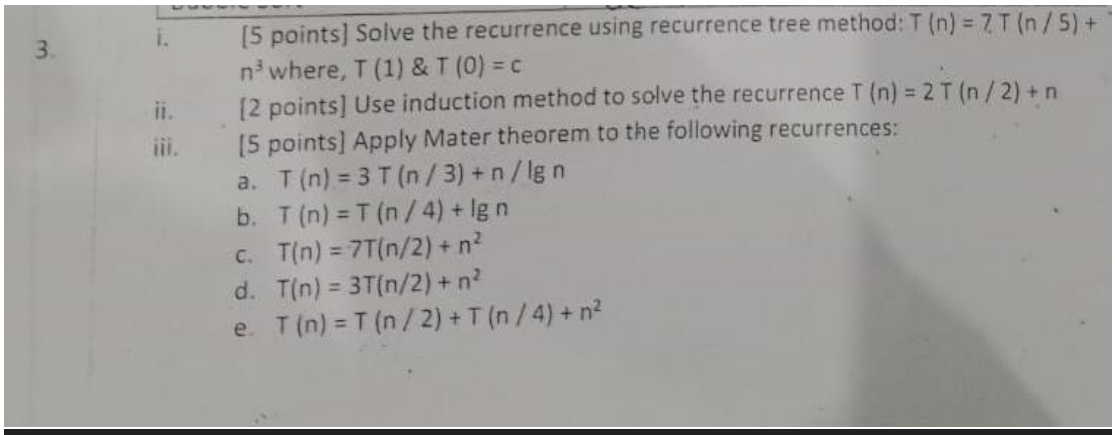
Algorithm:	Run Time	Efficiency class
Insertion Sort	Best = O(n) ; Avg, worst O(n²)	linear and polynomial
Merge Sort	Best, Avg, worst O(n log n)	logarithametic
Quick Sort	O(n²)	square polynomi
Strassen's Matrix Multiplication	O(n²)	polynomial, squ
Euclidean Algorithm	O(log n)	logarithmatic
Tower of Hanoi (Rec)	O(2ⁿ)	exponential
Binary Search	O(n) best = O(1)	linear
Bubble Sort	O(n²)	polynomial, squa

Algorithm	Best Case	Average Case	Worst Case	Efficiency Class
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Polynomial (Quadratic in worst)
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$n \log n$ ("Divide & Conquer")
Quick Sort (naïve pivot)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Mostly $n \log n$, worst n^2
Strassen's Matrix Multiply	—	—	$O(n^{\log_2 7}) \approx O(n^{2.8074})$	Polynomial (faster than n^3)
Euclidean Algorithm (GCD)	$O(\log n)$	$O(\log n)$	$O(\log n)$	Logarithmic
Tower of Hanoi (Recursive)	—	—	$O(2^n)$	Exponential
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	Logarithmic
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Polynomial (Quadratic in worst)

Short Takeaway

- **Insertion Sort / Bubble Sort** are $O(n^2)$ in the worst/average case.
- **Merge Sort** is $O(n \log n)$ in all cases.
- **Quick Sort** is $O(n \log n)$ on average, but $O(n^2)$ worst.
- **Strassen's Algorithm** is $\approx O(n^{2.8074})$.
- **Euclidean** is $O(\log n)$.
- **Tower of Hanoi** is $O(2^n)$.
- **Binary Search** is $O(\log n)$.

7



3.

i. [5 points] Solve the recurrence using recurrence tree method: T (n) = 7 T (n / 5) + n^3 where, T (1) & T (0) = c

ii. [2 points] Use induction method to solve the recurrence T (n) = 2 T (n / 2) + n

iii. [5 points] Apply Mater theorem to the following recurrences:

a. T (n) = 3 T (n / 3) + n / lg n
b. T (n) = T (n / 4) + lg n
c. T(n) = 7T(n/2) + n^2
d. T(n) = 3T(n/2) + n^2
e. T (n) = T (n / 2) + T (n / 4) + n^2

3.(i)

"Solve the recurrence using the recurrence-tree method:

$$T(n) = 7T\left(\tfrac{n}{5}\right) + n^3, \quad \text{with } T(1) = T(0) = c.$$

"

Solution (Recurrence-Tree or Master-Theorem Style)

- Here, $a = 7$, $b = 5$, and $f(n) = n^3$.
- The "critical exponent" from the Master theorem would be
$$n^{\log_b(a)} = n^{\log_5(7)} \approx n^{1.209}.$$
- We compare $f(n) = n^3$ to $n^{1.209}$. Clearly, n^3 grows faster:
$$n^3 = \Omega\left(n^{1.209+\varepsilon}\right) \quad (\text{with } \varepsilon \approx 1.79).$$
- By the usual Master-theorem "Case 3" (since n^3 dominates $n^{1.209}$ by a polynomial factor), we get
$$T(n) = \Theta\left(n^3\right).$$
- The base cases $T(1) = T(0) = c$ only shift the solution by a constant.
- **Answer:** $\boxed{T(n) = \Theta(n^3).}$

3.(i) Solve using the Recurrence Tree Method

$$T(n) = 7T(n/5) + n^3, \quad \text{where } T(1) \text{ and } T(0) = c.$$

Step 1: Expand the Recurrence

- We expand the recurrence in a tree form:

$$T(n) = 7T(n/5) + n^3$$

Expanding further:

$$T(n) = 7[7T(n/5^2) + (n/5)^3] + n^3$$
$$= 7^2T(n/5^2) + 7(n/5)^3 + n^3$$

Continuing expansion for k levels:

$$T(n) = 7^kT(n/5^k) + \sum_{i=0}^{k-1} 7^i(n/5^i)^3$$

When $n/5^k = 1$, we reach the base case. Solving for k:

$$n = 5^k \quad \Rightarrow \quad k = \log_5(n).$$

Step 2: Sum of Work Done at Each Level

Step 2: Sum of Work Done at Each Level

- The summation term simplifies to:

$$\sum_{i=0}^{\log_5(n)-1} 7^i \cdot n^3/5^{3i}.$$

- Since $7/5^3 < 1$, this forms a geometric series with sum:

$$O(n^3).$$

Final Answer

$$T(n) = \Theta(n^3).$$

3.(ii) Solve using Induction

$$T(n) = 2T(n/2) + n.$$

Step 1: Guess the Solution

We suspect $T(n) = O(n \log n)$. We prove this by **strong induction**.

Step 2: Base Case

For small n (e.g., $n = 1$), $T(1) = O(1)$, which holds.

Step 3: Inductive Step

Assume for $k < n$:

$$T(k) \leq ck \log k.$$

Now prove for n:

$$T(n) = 2T(n/2) + n.$$

Applying the induction hypothesis:

$$T(n) \leq 2 \left(c\frac{n}{2} \log \frac{n}{2} \right) + n.$$

Expanding:

$$= cn \log(n/2) + n$$

$$= cn(\log n - \log 2) + n$$

$$= cn \log n - cn + n.$$

For sufficiently large c, $-cn + n$ does not affect the asymptotic bound.

Final Answer

$$T(n) = \Theta(n \log n).$$

3.(iii) Apply the Master Theorem

(a) $T(n) = 3T(n/3) + n/\log n$

Step 1: Identify Master Theorem Terms

- $a = 3, b = 3, f(n) = n/\log n$.
- Critical exponent $n^{\log_3(3)} = n^1$.

Step 2: Compare $f(n)$ and n^1

- Since $f(n) = n/\log n$ is slightly smaller than n, we use the **Master Theorem extended case.**
- The result follows:

$$T(n) = \Theta(n \log\log n).$$

(b) $T(n) = T(n/4) + \log n$

Step 1: Identify Master Theorem Terms

- $a = 1, b = 4, f(n) = \log n$.
- Critical exponent $n^{\log_4(1)} = n^0 = 1$.

Step 2: Work Done in Each Level

The recursive depth is $\log_4 n$, and at each level, we have a cost of $\log n$. Summing over levels gives:

$$T(n) = O((\log n)^2).$$

(c) $T(n) = 7T(n/2) + n^2$

Step 1: Identify Master Theorem Terms

- $a = 7, b = 2, f(n) = n^2$.
- Critical exponent $n^{\log_2(7)} \approx n^{2.8}$.

Step 2: Compare $f(n)$ and $n^{2.8}$

Since n^2 is asymptotically smaller than $n^{2.8}$, we use **Case 1** of the Master theorem:

$$T(n) = \Theta(n^{\log_2(7)}).$$

(d) $T(n) = 3T(n/2) + n^2$

Step 1: Identify Master Theorem Terms

- $a = 3, b = 2, f(n) = n^2$.
- Critical exponent $n^{\log_2(3)} \approx n^{1.58}$.

Step 2: Compare $f(n)$ and $n^{1.58}$

Since n^2 is **larger** than $n^{1.58}$, we apply **Case 3** of the Master theorem:

$$T(n) = \Theta(n^2).$$

(e) $T(n) = T(n/2) + T(n/4) + n^2$

- This is not in **standard Master Theorem form**, but we compare with two recurrences:

1. **Upper bound:**

   $$T(n) \leq 2T(n/2) + n^2.$$

   Using Master theorem, this gives $\Theta(n^2)$.
2. **Lower bound:**

2. **Lower bound:**

$$T(n) \geq T(n/2) + n^2.$$

Using Master theorem, this also gives $\Theta(n^2)$.

- Since both bounds match, we conclude:

$$T(n) = \Theta(n^2).$$

Final Answers

1. **(i) Using Recurrence Tree**

$$T(n) = 7T(n/5) + n^3 \quad \Rightarrow \quad \Theta(n^3).$$

2. **(ii) Using Induction**

$$T(n) = 2T(n/2) + n \quad \Rightarrow \quad \Theta(n \log n).$$

3. **(iii) Applying Master Theorem**
   - (a) $T(n) = 3T(n/3) + n/\log n$
     $\Theta(n \log \log n)$
   - (b) $T(n) = T(n/4) + \log n$
     $\Theta((\log n)^2)$
   - (c) $T(n) = 7T(n/2) + n^2$
     $\Theta(n^{\log_2(7)})$
   - (d) $T(n) = 3T(n/2) + n^2$
     $\Theta(n^2)$
   - (e) $T(n) = T(n/2) + T(n/4) + n^2$
     $\Theta(n^2)$