SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 OF THE UGC ACT, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

Course

CSE318 - Algorithm Design Strategies & Analysis

Topic

Unit-1

Shri B. Srinivasan

Assistant Professor-III, School of Computing

SASTRA Deemed To Be University

Topics

- ✓ Algorithms Specification
  - ✓ Algorithm Design Model
  - ✓ Efficiency of Algorithms
- ✓ Asymptotic Notations
- ✓ Analysis of algorithms
  - ✓ Examples for complexity analysis
- ✓ Solving Recurrences
  - ✓ Recursion Tree Method
  - ✓ Iterative Method
  - ✓ Master Theorem
- ✓ Brute-force Method – Heuristic Method
  - ✓ Travelling Salesperson Problem
- ✓ Divide & Conquer Method
  - ✓ Maximum Sub Array
- ✓ Greedy Method
  - ✓ Knapsack Problem
  - ✓ Job Sequencing with Deadlines

Introducing Algorithm

What is an Algorithm?

- ✓ What is an Algorithm?
- ✓ Representation: Procedure, Flowchart and Pseudocode
- ✓ Why do we need many algorithms for solving same problem?
- ✓ How to choose best algorithm for a problem?
- ✓ How to quantify the Efficiency of an algorithm?
- ✓ What is "Analysis of Algorithm"?
- ✓ Time complexity and Space Complexity

Algorithm: Procedure vs Pseudocode

Procedure findMax

Input: Three distinct numbers, number1, number2 and number 3.

Output: The biggest number among the given 3 numbers

Steps:

1. Compare number1 with number2 and number3. If it is greater than both number2 and number3, return number1 as biggest.
2. Compare number2 with number1 and number3. If it is greater than both number1 and number3, return number2 as biggest.
3. Compare number3 with number1 and number2. If it is greater than both number2 and number3, return number1 as biggest.

Algorithm: Procedure vs Pseudocode

```
Algorithm findMax(a,b,c)
    Input: Three numbers: a, b and c
    Output: The biggest number
    if a>b and a>c then
            return a
    endif
    if b>a and b>c then
            return b
    endif
    if c>a and c>b then
            return c
    endif
end findMax
```

Pseudocode

```
int findMax(int a, int b, int c)
{
        if(a>b && a>c)
        {
                return a;
        }
        if(b>a && b>c)
        {
                return b;
        }
        if(c>a && c>b)
        {
                return c;
        }
}
```

Code in C Language

How to quantify Efficiency of an Algorithm?

Three different Algorithms for the same problem: *To find the maximum among 3 distinct numbers*

Which is the best?

```
int findMax(int a, int b, int c)
{
        if(a>b && a>c)
        {
                return a;
        }
        if(b>a && b>c)
        {
                return b;
        }
        if(c>a && c>b)
        {
                return c;
        }
}
```

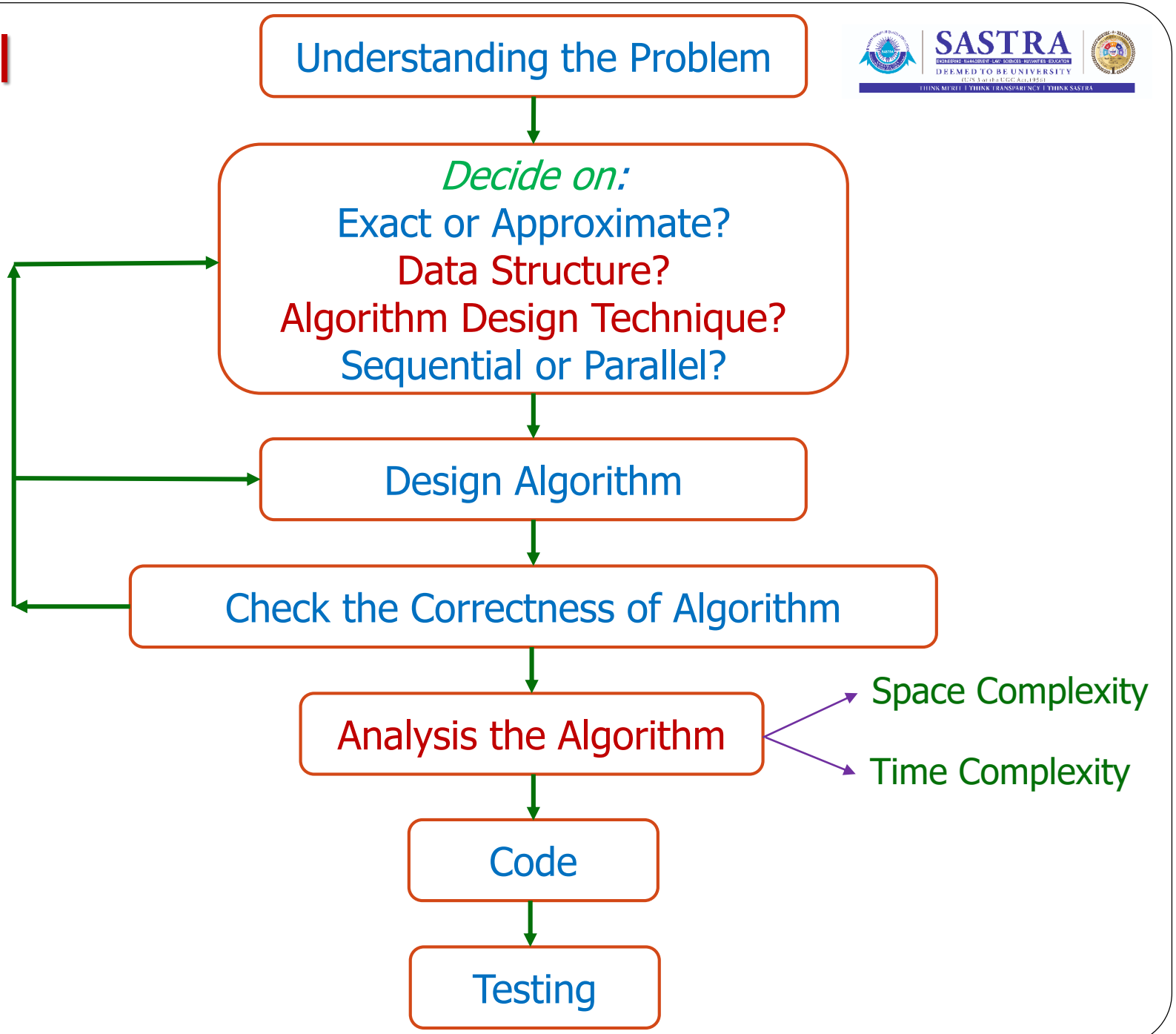
```
int findMax(int a, int b, int c)
{
        if(a>b && a>c)
        {
                return a;
        }
        else
        {
                if(b>c)
                {
                        return b;
                }
                else
                {
                        return c;
                }
        }
}
```

```
int findMax(int a, int b, int c)
{
        if(a>b)
        {
                if(a>c)
                {
                        return a;
                }
                else
                {
                        return c
                }
        }
        else
        {
                if(b>c)
                {
                        return b;
                }
                else
                {
                        return c;
                }
        }
}
```

Algorithm Design Model

Algorithm Design Model

1. What kind of Data Structure to be Choosed?
   - *Stack*
   - *Queue*
   - *List*
   - *Tree*
   - *Graph*
2. What kind of Algorithm Design Technique to be used to solve the problem?
   - *Brute-force*
   - *Divide & Conquer*
   - *Dynamic Programming*
   - *Greedy*
   - *Backtracking*
   - *Branch and Bound, etc.*
3. What is the Efficiency of algorithm? – Need to Analysis of Algorithm:
   - *Time Complexity*
   - *Space Complexity*



Asymptotic Notations – For Representing Time Complexity

Asymptotic Notations

- ✓ Mathematical Model – To represent the Time Complexity of an algorithm
- ✓ Time Complexity – Also known as Rate of Growth of an algorithm
- ✓ Asymptotic Notation – A relation between two function f(n) an g(n)
- ✓ It is a formal notation for discussing and analysing the "Classes of Functions"
- ✓ Example:

Let two functions, $f(n) = 4n^2+3n+5$ and $g(n) = n^2$

Then, $4n^2+3n+5 \in O(n^2)$

$4n^2+3n+5 \in \Omega(n^2)$

$4n^2+3n+5 \in \theta(n^2)$

Rate of Growth		
Input Size	Time	
n	Algorithm 1	Algorithm 2
1	1	1
10	20	10
100	5000	100
1000	500000	1000
Best?		✓

Asymptotic Notations

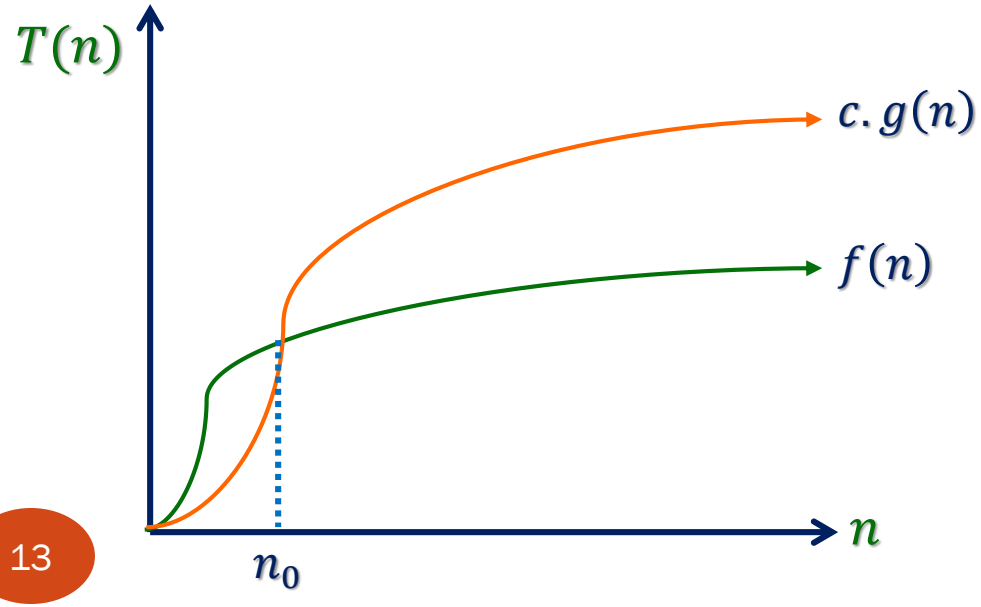
- ✓ Tight Bound
  - Big-Oh [O] – Define Upper Bound
  - Big-Omega [Ω] – Define Lower Bound
  - Theta [θ] – Define Both Upper and Lower Bound
- ✓ Loose Bound
  - Small-Oh or Little-Oh [o]
  - Small-Omega or Little-Omega [ω]

Note: Usually Big-Oh [O] and Theta [θ] notations are used for representing time complexity of an algorithm

Big-Oh [O]

A function f(n) is said to be in Big-Oh, O(g(n)), denoted as f(n) ∈ O(g(n)) or simply, f(n) = O(g(n)), if f(n) is bounded ABOVE by some +ve constant multiples of g(n) for all large 'n', if there exist some +ve constants c, and some non-negative integer n_0

$$0 \leq f(n) \leq c.g(n), \forall n \geq n_0$$



Example

Let two functions, $f(n) = 4n^2+3n+5$ and $g(n) = n^2$, Then, prove that $4n^2+3n+5 \in O(n^2)$

Solution:

- Let $f(n) = 4n^2+3n+5$.
- Rise the terms '3n' and '5' to n^2
- Need to multiply by n and n^2 respectively
- $f(n) = 4n^2+3n+5 \leq 4n^2+3n^2+5n^2$
- $f(n) \leq 4n^2+3n^2+5n^2 \rightarrow f(n) \leq 12n^2$
- $f(n) \leq 12.\ g(n)$, where c = 12.
- Need to prove this is true for large n values.
- Need to verify for n=1,2,3, and so on.,

n=1 → $4(1)^2+3(1)+5 \leq 12.\ (1)^2$ → $12 \leq 12$ ✓

n=2 → $4(2)^2+3(2)+5 \leq 12.\ (2)^2$ → $27 \leq 48$ ✓

n=3 → $4(3)^2+3(3)+5 \leq 12.\ (3)^2$ → $50 \leq 108$ ✓

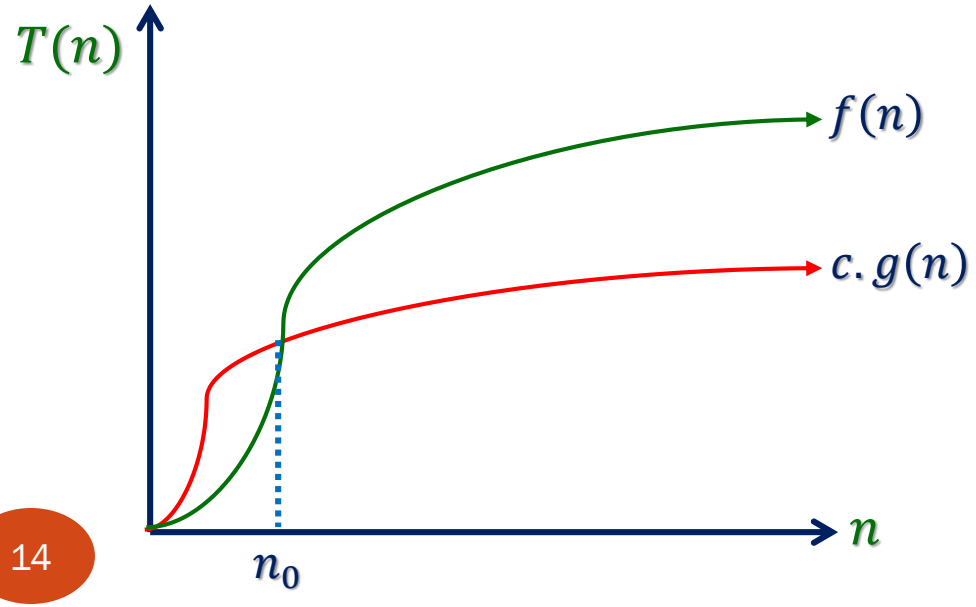
So, $f(n) \leq c.g(n)$ is True for c=12 and n_0=1

Hence proved, $4n^2+3n+5 \in O(n^2)$

Big-Omega [Ω]

A function f(n) is said to be in Big-Omega, Ω(g(n)), denoted as f(n) ∈ Ω(g(n)) or simply, f(n) = Ω(g(n)), if f(n) is bounded BELOW by some +ve constant multiples of g(n) for all large 'n', if there exist some +ve constants c, and some non-negative integer n_0

$$0 \geq f(n) \geq c.g(n), \forall n \geq n_0$$



Example

Let two functions, $f(n) = 4n^2+3n+5$ and $g(n) = n^2$, Then, prove that $4n^2+3n+5 \in \Omega(n^2)$

Solution:

- Let $f(n) = 4n^2+3n+5$.
- Reduce the terms '3n' and '5' to 0
- Need to remove these terms, so we can write,
- $f(n) = 4n^2+3n+5 \geq 4n^2$
- $f(n) \geq 4n^2 \rightarrow f(n) \geq 4.\ g(n)$, where c = 4.
- Need to prove this is true for large n values.
- Need to verify for n=1,2,3, and so on.,

n=1 → $4(1)^2+3(1)+5 \geq 4.\ (1)^2$ → $12 \geq 4$ ✓

n=2 → $4(2)^2+3(2)+5 \geq 4.\ (2)^2$ → $27 \geq 16$ ✓

n=3 → $4(3)^2+3(3)+5 \geq 4.\ (3)^2$ → $50 \geq 36$ ✓

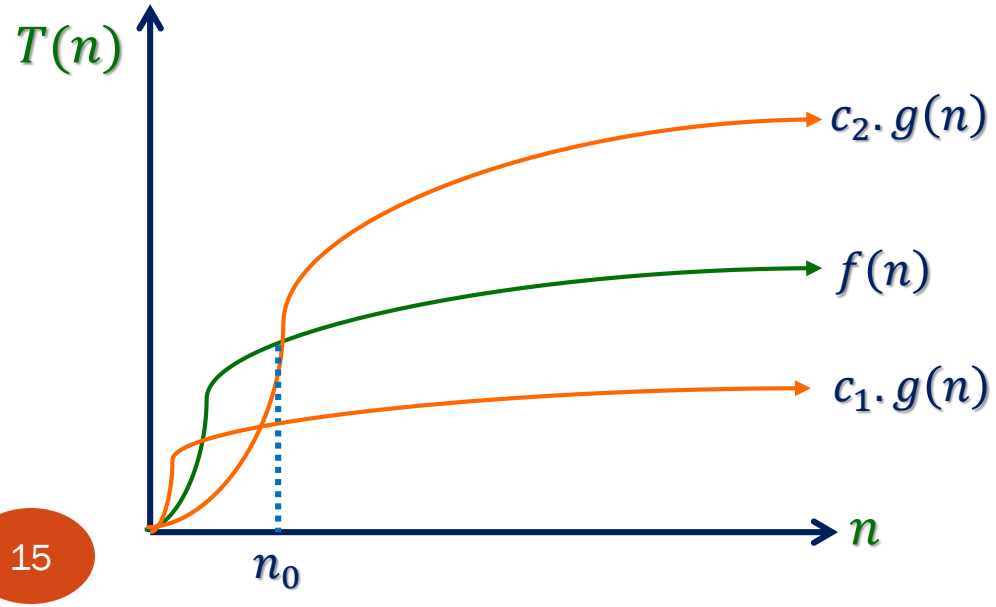
So, $f(n) \geq c.g(n)$ is True for c=4 and n_0=1

Hence proved, $4n^2+3n+5 \in \Omega(n^2)$

Theta [θ]

A function f(n) is said to be in Theta, θ(g(n)), denoted as f(n) ∈ θ(g(n)) or simply, f(n) = θ(g(n)), if f(n) is bounded BELOW and ABOVE by some +ve constant multiples of g(n) for all large 'n', if there exist some +ve constants c_1 and c_2, and some non-negative integer n_0

$$c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n \geq n_0$$



Example

Let two functions, f(n) = $4n^2+3n+5$ and g(n) = n^2 .
Then, prove that $4n^2+3n+5$ ∈ θ (n^2)

Solution:

- If f(n) ∈ O(g(n)) and f(n) ∈ Ω(g(n)), then f(n) ∈ θ(g(n))
- We already proved f(n) ∈ O(g(n)) and f(n) ∈ Ω(g(n))
- So, f(n) ∈ θ(g(n))

So, $c_1.g(n) \leq f(n) \geq c_2.g(n)$ is True for

$c_1=4$, $c_2=12$ and $n_0=1$

Hence proved, $4n^2+3n+5$ ∈ Ω(n^2)

Small-Oh or Little-Oh [o]

A function f(n) is said to be in Small-Oh, o(g(n)), denoted as f(n) ∈ o(g(n)) or simply, f(n) = o(g(n)), if

$$0 \leq f(n) < c.g(n), \forall n \geq n_0$$

Small-Omega or Little-Omega [ω]

A function f(n) is said to be in Small-Omega, ω(g(n)), denoted as f(n) ∈ ω(g(n)) or simply, f(n) = ω(g(n)), if

$$0 \geq f(n) > c.g(n), \forall n \geq n_0$$

Rate of Growth – Comparing Algorithms

Rate of Growth

- ✓ Constant – $O(1)$ *or* $O(c)$
- ✓ Linear – $O(n)$
- ✓ Quadratic – $O(n^2)$
- ✓ Cubic – $O(n^3)$
- ✓ Polynomial – $O(n^d),\ d > 1$
- ✓ Logarithmic – $O(\log n)$
- ✓ Poly-logarithmic - $O(n^d \log n),\ d \geq 1$
- ✓ Fractional Power – $O(n^c),\ 0 < c < 1$
- ✓ Constant Power – $O(c^n),\ \ c > 1, 2^n, 3^n, 4^n, \ldots$
- ✓ Factorial – $O(n!)$

How to Compare Algorithms?

- ✓ If the time complexity of algorithm is given, we can rank the algorithm based on their efficiency in terms of time complexity.
- ✓ Substitute some large n value. Based on result we can rank.

Example:

Algorithm	Time Complexity	Input Size - n			Rank
		8	256	1000	
Algorithm 1	O(n)	8	256	1000	4
Algorithm 2	O(log n)	3	8	9.96578428	2
Algorithm 3	$O(n^2)$	64	65536	1000000	6
Algorithm 4	O(n log n)	24	2048	9965.78428	5
Algorithm 5	O(n!)	40320	Infinity	Infinity	7
Algorithm 6	O(1)	1	1	1	1
Algorithm 7	$O(\sqrt{n})$	2.82843	16	31.6227766	3

Problems

1. Show that $n^2+2n+1 \in O(n^2)$
2. Show that $3n+1 \in O(n)$
3. Show that $2^{n+1} \in O(3^n)$
4. Show that $3n^2+n \lg n \in O(n^2)$

1. Show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$

Solution:

$f(n) = n^2 + 2n + 1$

$g(n) = O(n^2)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{n^2 + 2n + 1}{n^2} < \frac{n^2 + 2n^2 + n^2}{n^2}$$

$$n = 1 = \frac{1 + 2 + 1}{1} = 4$$

Choose $c = 4$

$$2n < 2n^2$$

Thus $n^2 + 2n + 1$ is $O(n^2)$ because $n^2 + 2n + 1 \leq 4n^2$ $\forall n$

2. Show that $f(n) = 3n+7$ is $O(n)$

solution:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n+7}{n} = \frac{3n+7n}{n} = \frac{10n}{n} = 10$$

choose $c = 10$

$7 < 7n$

Thus $3n+7$ is $O(n)$ because $3n+7 \leq 10n$ whenever $n \geq 1$

3. Prove that

$$f(n) = 2^{n+1} \in O(3^n)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{2^{n+1}}{3^n} = \frac{2^n \cdot 2^1}{3^n} = 2\left(\frac{2}{3}\right)^n = 2\left(\frac{2}{3}\right)^{\infty}$$

$$\left(\frac{2}{3}\right)^{\infty} \to 0$$

Hence we get, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \in O(3^n)$

1. Prove that

$$3n^2 + n\lg n \in \Theta(n^2)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n^2 + n\lg n}{n^2}$$

$$= \lim_{n\to\infty} \frac{3n^2}{n^2} + \lim_{n\to\infty} \frac{n\lg n}{n^2}$$

$$= \lim_{n\to\infty} \frac{\lg n}{n} = 0$$

$$= 3 + 0 = 3 \text{ (Some constant)}$$

$$\therefore \text{it means that } \in \Theta(n^2)$$

Analyzing Algorithms - Examples

How to Analyze Algorithm?

- ✓ Time Complexity – Rate of Growth of an algorithm
- ✓ It depends on "*How many operations are going to be executed*?"
- ✓ So, we need to count the number of operations to be performed.
- ✓ This count is usually depends on the input size 'n' for many algorithms.
- ✓ By counting the number of operations, it will give an polynomial expression in terms of input size n.
- ✓ Then, this polynomial function will be expressed as Big-Oh or Theta classes of functions.
- ✓ For example, if we get $4n^2+3n+5$ as the count, we can say that, this function is belongs to $O(n^2)$

Example:

Sum of N numbers

```
Algorithm Sum(A[1..n])
        S ← 0                          ----------> 1
        For i←1 to n do                ----------> (n-1+1)+1 = n+1
                S ← S + A[i]           ----------> 2n
        End For
        return S                       ----------> 1
End Sum
```

$$T(n) = 1 + n + 1 + 2n + 1 = 3n + 3 \in O(n)$$

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

Note: To find the number of iterations in a 'for' loop, we may use the formula, upper-lower+1

Some Observations...

- ✓ To find time complexity, we need to *count the number of operations*
- ✓ The number of operations mainly depends on the ACTIVE operations.
- ✓ Active operation is an operation which is executed more often in an algorithm
- ✓ So, most probably, the active operation is operation present in the iteration.
- ✓ So, we may count the number of iterations to be executed, if the algorithm containing iterative statements.
- ✓ Mostly, the number of iterations are depends on input size.
- ✓ So, the time complexity is represented in terms of input size.

Problem 1

```
for(i=1; i<=n; i++)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \sum_{i=1}^{n} 1 = n - 1 + 1 = n \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 2

```
for(i=n; i>=1; i--)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \sum_{i=1}^{n} 1 = n - 1 + 1 = n \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 3

```
for(i=1; i<=n; i=i+2)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \frac{\sum_{i=1}^{n} 1}{2} = \frac{n-1+1}{2} = \frac{n}{2} \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 4

```
for(i=1; i<=n; i=i*2)
{
        O(1) Statements…
}
```

Solution

$$T(n) = [i \leq n] = [2^{k-1} \leq n] = [k - 1 \leq \log_2 n] \Rightarrow k = \log_2 n + 1$$

Iteration	i
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
.	
.	
k	2^{k-1}

$$T(n) = O(\log_2 n)$$

Problem 5

```
for(i=n; i>=1; i=i/2)
{
        O(1) Statements…
}
```

Solution

$$T(n) = [i \geq 1] = \left[\frac{n}{2^{k-1}} \geq 1\right] = [n \geq 2^{k-1}] = [\log_2 n \geq k-1]$$

$$T(n) = [k-1 \leq \log_2 n] \Rightarrow k = \log_2 n + 1$$

Iteration	i
1	$n = n / 2^0$
2	$n/2 = n / 2^1$
3	$n/4 = n / 2^2$
4	$n/8 = n / 2^3$
.	
.	
k	$n / 2^{k-1}$

$$T(n) = O(\log_2 n)$$

Problem 6

```
for(i=1, P=1; P<=n; i++)
{
        P = P + i;
}
```

Solution

$$T(n) = [P \leq n] = [k^2 \leq n] = [k \leq \sqrt{n}] \Rightarrow k \in O(\sqrt{n})$$

Iteration	P=P+i
1	1 = 1
2	3 = 1 + 2
3	6 = 1 + 2 + 3
4	10 = 1 + 2 + 3 + 4
.	
.	
k	1+2+3+...+k = k(k+1)/2 = O(k^2)

$$T(n) = O(\sqrt{n})$$

Problem 7

```
for(i=1; i*i<=n; i++)
{
        O(1) Statements…
}
```

Solution

$$T(n) = [i^2 \leq n] = [k^2 \leq n] = [k \leq \sqrt{n}] \Rightarrow k \in O(\sqrt{n})$$

Iteration	i*i
1	1
2	$2*2 = 2^2$
3	$3*3 = 3^2$
4	$4*4 = 4^2$
.	
.	
k	$k*k = k^2$

$$\boldsymbol{T(n) = O(\sqrt{n})}$$

Problem 8

```
for(i=1; i<=n; i++)
{
        O(1) Statements…
}
for(j=1; j<=n; j++)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \sum_{i=1}^{n} 1 + \sum_{j=1}^{n} 1$$

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

$$T(n) = 2n \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 9

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=n; j++)
        {
                O(1) Statements…
        }
}
```

Solution

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n - 1 + 1$$

$$T(n) = \sum_{i=1}^{n} n = n \,.\sum_{i=1}^{n} 1$$

$$T(n) = n\,.(n - 1 + 1) = n * n$$

$$T(n) = n * n \in O(n^2)$$

$$\boldsymbol{T(n) = O(n^2)}$$

Problem 10

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=i; j++)
        {
                O(1) Statements…
        }
}
```

Solution

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i - 1 + 1$$

$$T(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots n$$

$$T(n) = n \cdot \frac{(n+1)}{2}$$

$$T(n) = \frac{1}{2}(n^2 + n) \in O(n^2)$$

$$\boldsymbol{T(n) = O(n^2)}$$

Problem 11

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=n; j++)
        {
                for(k=1;k<n;k++)
                {
                        O(1) Statements…
                }
        }
}
```

Solution

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

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} n = n \,.\sum_{i=1}^{n}\sum_{j=1}^{n} 1$$

$$T(n) = n \,.\sum_{i=1}^{n} n = n.n.\sum_{i=1}^{n} 1 \,.= \; n^3$$

$$T(n) = n * n * n \in O(n^3)$$

$$\boldsymbol{T(n) = O(n^3)}$$

How to analyze algorithm?

If the number of Iterations depends on
Not only the input size, but also the input values

Example: Linear Search

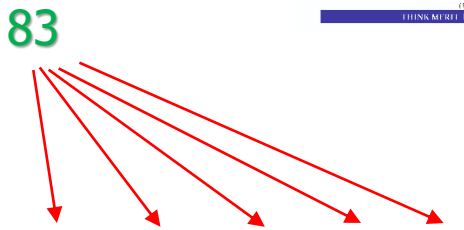
Example:

Linear Search

```
Algorithm Search(A[0..n-1], n,  Key)
        For i←0 to n-1 do
                If Key = A[i] Then
                        Return True
                End IF
        End For
        Return False
End Search
```

- ✓ In this example, the exact number of iterations is not known in advance.
- ✓ It depends on, "where the key element is present in the array?"

Example:



Consider an array A = [29, 45, 12, 57, 83]

Case 1: [If Key is 29]

In this case, since the key is present in the first location, the algorithm need only one iteration to give the result.

Case 2: [If Key is 83 OR 25]

If key is present in last location OR the key is not present in the array.

In this case, the algorithm needs 'n' iterations to obtain the result.

Worst, Best and Average Case Complexity

When the number of iterations depends on value of the input, we need to analyse the algorithm in three different cases.

Best Case Complexity: [Case 1 in above example]

It is the time complexity of the algorithm for the BEST CASE INPUT.

Worst Case Complexity: [Case 2 in above example]

It is the time complexity of the algorithm for the WORST CASE INPUT.

Average Case Complexity:

This can be obtained by considering probability of input elements.

For Linear Search Algorithm:

$$T_{Best}(n) = O(1)$$

$$T_{Worst}(n) = O(n)$$

$$T_{Average}(n) = O(n)$$

Insertion Sort – Algorithm, Example & Analysis

Insertion Sort – Algorithm

```
Algorithm InsertionSort(a[0..n-1], n)
        for i=1 to n-1 do
                Key = a[i]
                j = i-1
                while j>=0 and Key<a[j] do
                        a[j+1] = a[j]
                        j = j-1
                end while
                a[j+1] = Key
        end for
end InsertionSort
```

Insertion Sort – Example (Tracing Algorithm)

Iteration-1 (i=1)

Key = a[i]

Key
23

		j=0	j=-1	a[j+1] = Key
0	56	56	Stop	23
1	23	56		56
2	76	76		76
3	34	34		34
4	65	65		65
5	87	87		87
6	20	20		20
7	45	45		45

Iteration-2 (i=2)

Key

Key
76

76<56, false, stop the iteration

		j=1	j=0	j=-1	a[j+1] = Key
0	23	23		Stop	23
1	56	56			56
2	76	76			76
3	34	34			34
4	65	65			65
5	87	87			87
6	20	20			20
7	45	45			45

Stop

Insertion Sort – Example (Tracing Algorithm)

Key

34

Iteration-3 (i=3)

34<76 34<56 34<23

	j=2	j=1	j=0	j=-1
0	23	23	23	23
1	56	56	56	56
2	76	76	56	56
3	34	76	76	76
4	65	65	65	65
5	87	87	87	87
6	20	20	20	20
7	45	45	45	45
				Stop

Stop

a[j+1] = Key

23
34
56
76
65
87
20
45

Insertion Sort – Example (Tracing Algorithm)

Key

65

Iteration-4 (i=4)

65<76 65<56

		j=3	j=2	j=1	j=0	j=-1
0	23	23	23			Stop
1	34	34	34			
2	56	56	56			
3	76	76	76			
4	65	76	76			
5	87	87	87			
6	20	20	20			
7	45	45	45			
			Stop			

a[j+1] = Key

23
34
56
65
76
87
20
45

Insertion Sort – Example (Tracing Algorithm)

Key

87

Iteration-5 (i=5)

87<76

		j=4	j=3	j=2	j=1	j=0	j=-1
0	23	23					Stop
1	34	34					
2	56	56					
3	65	65					
4	76	76					
5	87	87					
6	20	20					
7	45	45					
		Stop					

a[j+1] = Key

23	0
34	1
56	2
65	3
76	4
87	5
20	6
45	7

Insertion Sort – Example (Tracing Algorithm)

Iteration-6 (i=6)

Key: 20

20<87 20<76 20<65 20<56 20<34 20<23

j=5	j=4	j=3	j=2	j=1	j=0	j=-1
23	23	23	23	23	23	23	Stop
34	34	34	34	34	34	23	
56	56	56	56	56	34	34	
65	65	65	65	56	56	56	
76	76	76	65	65	65	65	
87	87	76	76	76	76	76	
20	87	87	87	87	87	87	
45	45	45	45	45	45	45	

a[j+1] = Key

a[j+1] = Key
20
23
34
56
65
76
87
45

Insertion Sort – Example (Tracing Algorithm)

Key = a[i]

Iteration-7 (i=7) | 45

45<87 45<76 45<65 45<56 45<34

	j=6	j=5	j=4	j=3	j=2	j=1	j=0	j=-1	a[j+1] = Key
0	20	20	20	20	20	20			20
1	23	23	23	23	23	23			23
2	34	34	34	34	34	34			34
3	56	56	56	56	56	56			45
4	65	65	65	65	56	56			56
5	76	76	76	65	65	65			65
6	87	87	76	76	76	76			76
7	45	87	87	87	87	87			87
					Stop			Stop	

Insertion Sort – Analysis of Algorithm

```
Algorithm InsertionSort(a[0..n-1], n)
        for i=1 to n-1 do                                   → 1. Outer Loop
            1   Key = a[i]
            2   j = i-1
                while j>=0 and Key<a[j] do                  → 2. Inner Loop
                    3   a[j+1] = a[j]
                    4   j = j-1
                end while
            5   a[j+1] = Key
        end for
end InsertionSort
```

Insertion Sort – Analysis of Algorithm

1. Outer Loop
   - Since the loop is 'for' loop, we can find the exact number of iterations for this loop.
   - This executes for i=1 to n-1
   - Number of Iterations: (n-1)-1+1 = n-1 [upper-lower+1]
2. Inner Loop
   - Number of iterations for the 'while' loop is not known in advance.
   - It is depending on the condition Key<a[j]
   - It means, the number of iterations is depending on the Input Values a[j] and Key
   - So, it varies for each outer iteration (each 'i' value)
   - Let us cosider,
     - t_1 be the number of 'while' loop iterations for i=1
     - t_2 be the number of 'while' loop iterations for i=2
     - t_3 be the number of 'while' loop iterations for i=3
     - ….
     - t_{n-1} be the number of 'while' loop iterations for i=n-1

Insertion Sort – Analysis of Algorithm

- So, the total number of 'while' loop iterations is expressed as:

$$t_1 + t_2 + t_3 + \ldots\ldots\ldots\ldots + t_{n-1}$$

- It can be simply written as:

$$\sum_{i=1}^{n-1} t_i$$

- The statements (outer loop), 1, 2 and 5 are executed for $n - 1$ times.
- The statements (inner loop), 3 and 4 are executed for $\sum_{i=1}^{n-1} t_i$
- So, the running time of the Insertion Sort is expressed as:

$$T(n) = (n-1) + \sum_{i=1}^{n-1} t_i$$

$$T(n) = O(n) + \sum_{i=1}^{n-1} t_i$$

Insertion Sort – Analysis of Algorithm

$$T(n) = O(n) + \sum_{i=1}^{n-1} t_i$$

- Now, t_i value is depending on the input values
- So, the algorithm is now needs to be analysed for different cases of input.
- Usually, there are three cases of inputs, "Best Case", "Worst Case" and "Average Case".
- The time complexity will be analysed as "Best Case Complexity", "Worst Case Complexity" and "Average Case complexity"
- Best Case Complexity – Time complexity of an algorithm for best case input.
- Worst Case Complexity – Time complexity of an algorithm for worst case input. This case analyse the performance of the algorithm when the input is not favour.

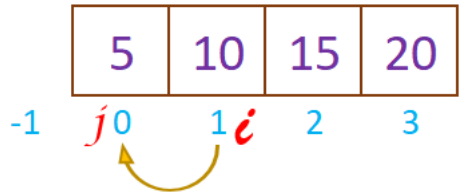
Insertion Sort – Analysis of Algorithm

Best Case Input

- Consider the following sample input.
- The input is already in ascending order

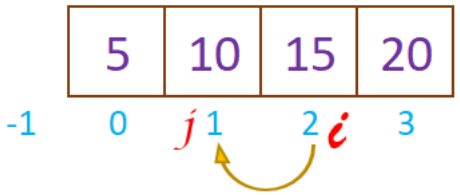
$$T(n) = O(n) + \sum_{i=1}^{n-1} t_i$$

$i = 1$



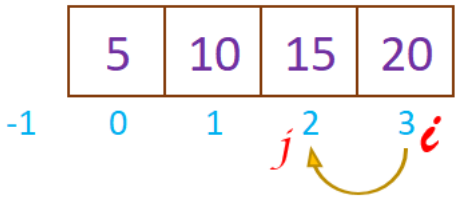
$t_1 = 1$

$i = 2$



$t_2 = 1$

$i = 3$



$t_3 = 1$

- So, if the input is already in order, the number of 'while' loop iterations always 1.
- $t_1 = t_2 = t_3 = \ldots\ldots = t_{n-1} = 1$
- Simply, $t_i = 1$
- Substitute this in the equation.

$$T(n) = O(n) + \sum_{i=1}^{n-1} 1$$

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

$$T_{best}(n) = O(n)$$

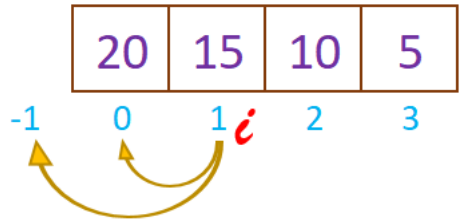
Insertion Sort – Analysis of Algorithm

Worst Case Input

- Consider the following sample input.
- The input is completely in Reverse Order

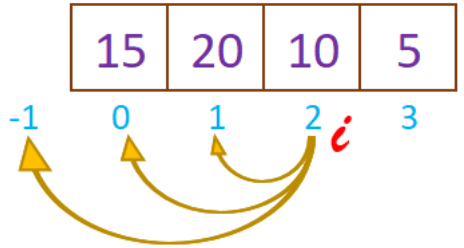
$$T(n) = O(n) + \sum_{i=1}^{n-1} t_i$$

$i = 1$



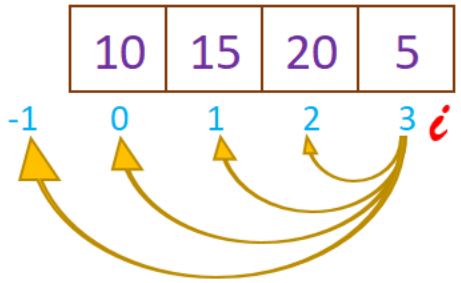
$t_1 = 2$

$i = 2$



$t_2 = 3$

$i = 3$



$t_3 = 4$

- So, if the input is in reverse order, the number of 'while' loop iterations at the i^{th} iteration is i-1.
- Simply, $t_i = i + 1$
- Substitute this in the equation.

$$T(n) = O(n) + \sum_{i=1}^{n-1} i + 1$$

$$T(n) = O(n) + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1$$

$$T(n) = O(n) + O(n^2) + O(n) = O(n^2)$$

Formula Used:

$$\sum_{i=1}^{n} i^d = O(n^{d+1})$$

$$T_{worst}(n) = O(n^2)$$

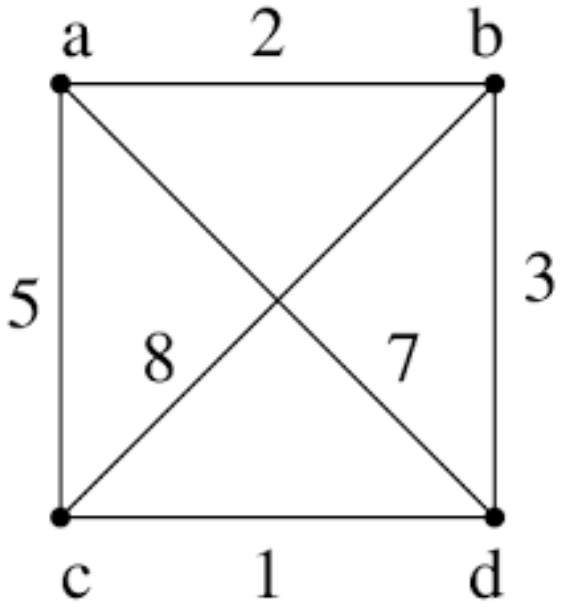
Brute-Force Method – Heuristic Approach

Brute-Force Method – Heuristic Approach

- ✓ A heuristic is a mental shortcut commonly used to simplify problems and avoid cognitive overload. Heuristics are part of how the human brain evolved and is wired, allowing individuals to quickly reach reasonable conclusions or solutions to complex problems.
- ✓ A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem.
- ✓ This is the most basic and simplest type of algorithm.
- ✓ More technically it is just like iterating every possibility available to solve that problem.
- ✓ **Example:** If there is a lock of 4-digit PIN. The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 (10^4) tries to find the right combination

Travelling Salesperson Problem (TSP) – Heuristic Approach

- ✓ Given a set of cities and distance between each pair of cities.
- ✓ The problem is to find the shortest possible trip to visit every city exactly once and returns to the starting city.
- ✓ The cities are mapped with the vertices and the distance between the pairs are mapped with the edge cost of a graph.
- ✓ TSP - Heuristic method



- ✓ Assume the City 'a' is the starting city
- ✓ The following trips are possible

Trip	Cost	Total Cost
a b c d a	2 + 8 + 1 + 7	18
a b d c a	2 + 3 + 1 + 5	11
a c b d a	5 + 8 + 3 + 7	23
a c d b a	5 + 1 + 3 + 2	11
a d b c a	7 + 3 + 8 + 5	23
a d c b a	7 + 1 + 8 + 2	18

- ✓ The Trip with minimum cost 11 is selected

Travelling Salesperson Problem (TSP) – Heuristic Approach

Algorithm TSP(Cost[1..n][1..n], n, S)

Input: n – Number of Vertices
S – Starting Vertex
Cost[1..n][1..n] - 'n x n' Cost matrix.

Output: Shortest Tour and the Cost

//Store all the vertices except the starting vertex into a list

For i←1 to n ***Do***
    If i<>S ***Then***
        Vertex[i] ← i
    End If
End For

//Let the initial minPath as infinity

minPath ← ∞

//Repeat until there is no other permutations

While NextPermutation(Vertex, n) ***Do***
    currentPathWeight ← 0
    K ← S
    For i←1 to n ***Do***
        currentPathWeight = currentPathWeight + Cost[K][Vertex[i]]
        K ← Vertex[i]
    End For
    currentPathWeight = currentPathWeight + Cost[K][S]
    minPath = Min(minPath, currentPathWeight)
End While
Return minPath

End TSP

Maximum Sub Array Problem

Maximum Sub Array Problem:

Input : An array of elements : +Ve & -Ve

Output : A sub array of input array whose Sum is maximum

Ex: I/P: $\overset{1}{[}-1, \overset{2}{3}, \overset{3}{-2}, \overset{4}{5}]$ ⇒ [-1] [3] [-2] [5] ⇒ l=1

[-1,3] [3,-2], [-2,5] ⇒ l=2

[-1, 3, -2], [3, -2, 5] ⇒ l=3

[-1, 3, -2, 5] ⇒ l=4

(6) ✓

⇓

Output

Algorithms:

1. Bruteforce ⇒ $O(n^3)$
2. Bruteforce ⇒ $O(n^2)$
3. Divide & Conquer ⇒ $O(n \log n)$
4. $O(n)$ Algorithm.

$O(n^3)$ Algorithm for Max. Subarray :

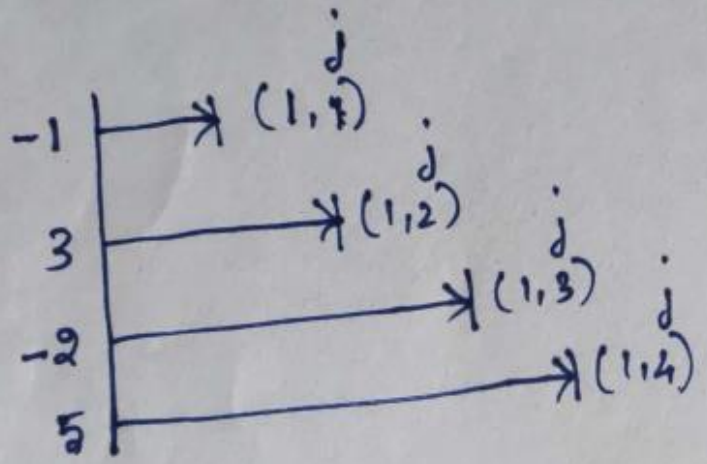
	mRange	mSum
	(2, 4)	6

	1	2	3	4
A	-1	3	-2	5

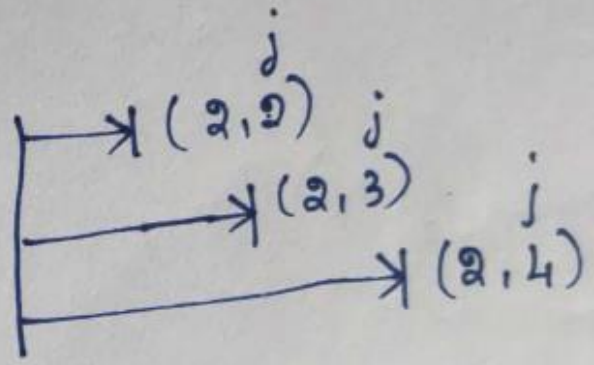
(1,1) ⇒ [-1]

(1,3) ⇒ [-1, 3, -2]

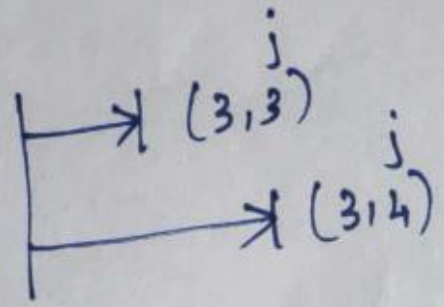
i = 1



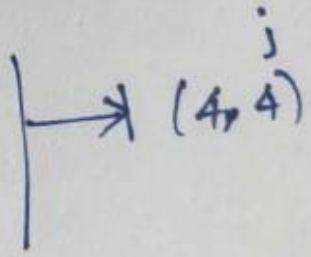
i = 2

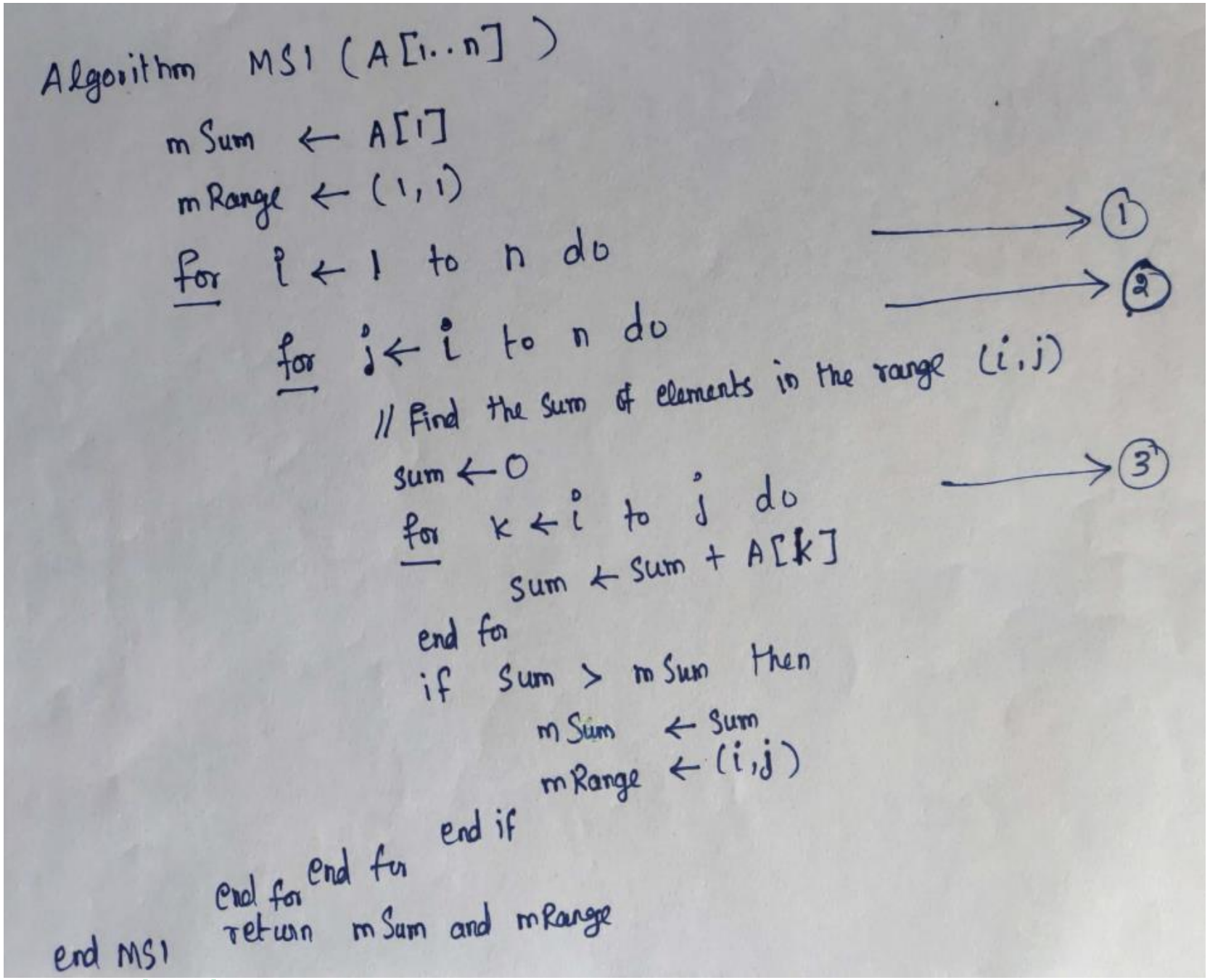


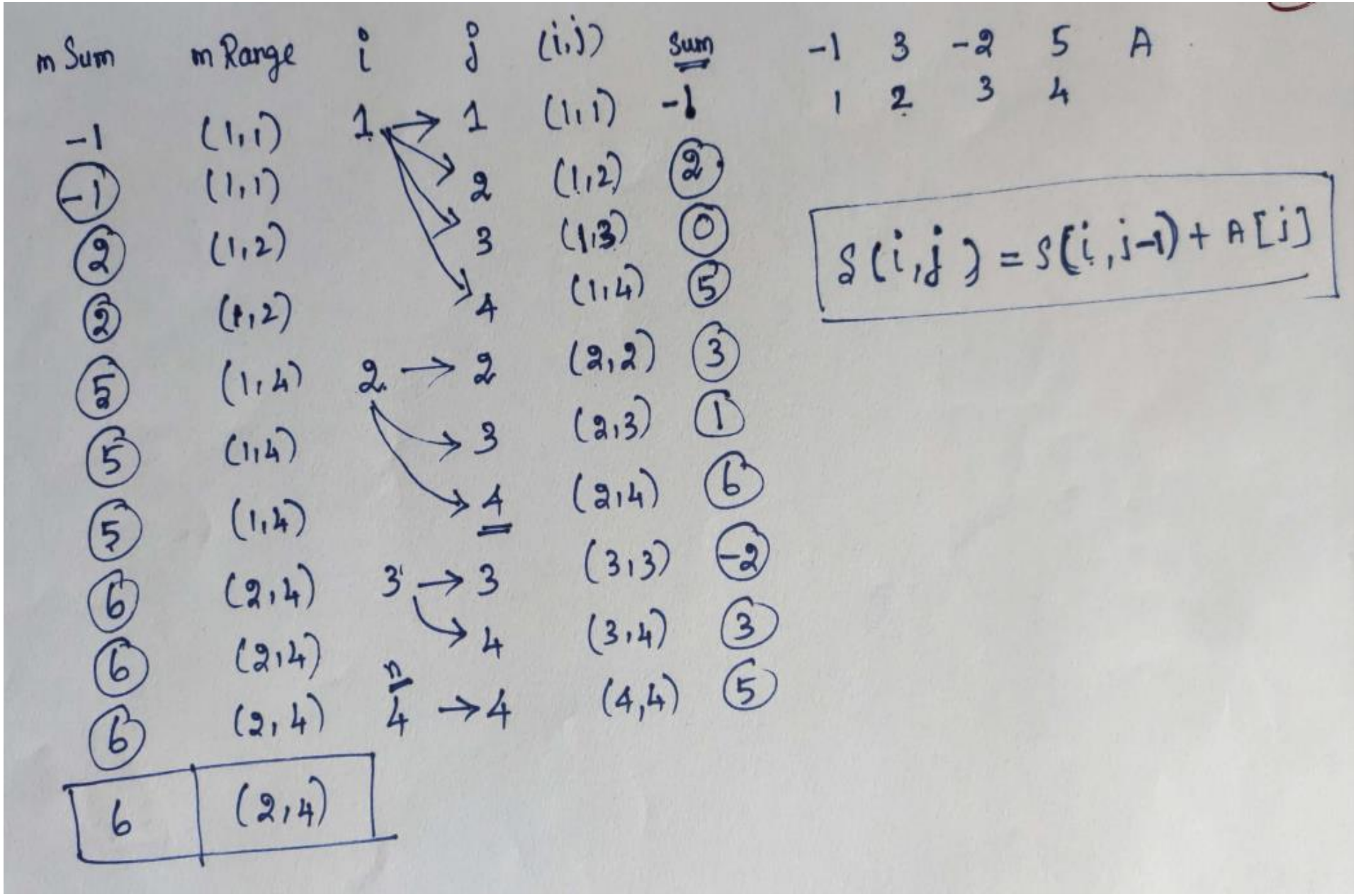
i = 3

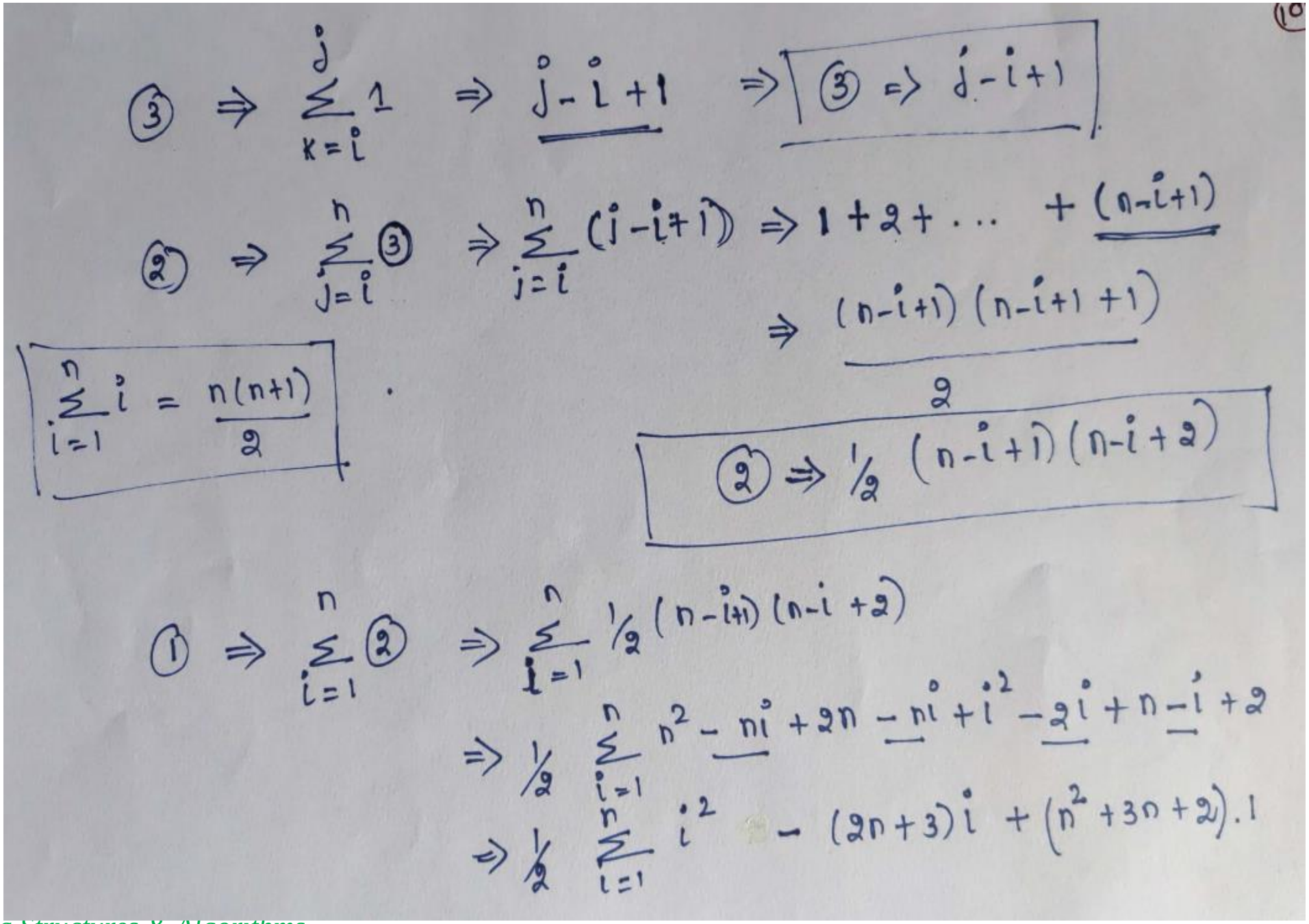


i = 4




Algorithm MS1 (A[1..n])
mSum ← A[1]
mRange ← (1,1)
for i ← 1 to n do
for j ← i to n do
// Find the sum of elements in the range (i,j)
sum ← 0
for k ← i to j do
sum ← sum + A[k]
end for
if sum > mSum then
mSum ← sum
mRange ← (i,j)
end if
end for
end for
return mSum and mRange
end MS1
1
2
3


mSum
mRange
i
j
(i,j)
Sum
-1 3 -2 5 A
1 2 3 4
S(i,j) = S(i,j-1) + A[j]
6
(2,4)


③ ⇒ $\sum_{k=i}^{j} 1$ ⇒ $j - i + 1$ ⇒ ③ ⇒ $j - i + 1$
② ⇒ $\sum_{j=i}^{n}$ ③ ⇒ $\sum_{j=i}^{n} (j - i + 1)$ ⇒ $1 + 2 + \ldots + (n - i + 1)$
⇒ $\frac{(n-i+1)(n-i+1+1)}{2}$
$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
② ⇒ $\frac{1}{2}(n-i+1)(n-i+2)$
① ⇒ $\sum_{i=1}^{n}$ ② ⇒ $\sum_{i=1}^{n} \frac{1}{2}(n-i+1)(n-i+2)$
⇒ $\frac{1}{2} \sum_{i=1}^{n} n^2 - ni + 2n - ni + i^2 - 2i + n - i + 2$
⇒ $\frac{1}{2} \sum_{i=1}^{n} i^2 - (2n+3)i + (n^2+3n+2).1$

$$\Rightarrow \frac{1}{2}\left[\sum_{i=1}^{n} i^2 - (2n+3)\cdot\sum_{i=1}^{n} i + (n^2+3n+2)\cdot\sum_{i=1}^{n} 1\right]$$

$$\Rightarrow \frac{1}{2}\left[O(n^3) - (2n+3)\cdot O(n^2) + (n^2+3n+2)O(n)\right]$$

$$\Rightarrow \frac{1}{2}\left[O(n^3) + O(n^3) + O(n^2) + O(n)\right] \in O(n^3)$$

$$T(n) = O(n^3)$$

$$\sum_{i=1}^{n} 1 = n-1+1 \Rightarrow n \Rightarrow O(n)$$

① $$\sum_{i=l}^{h} 1 = h-l+1$$

$$\sum_{i=1}^{n} i = 1+2+3+\ldots+n = \frac{n(n+1)}{2} \in O(n^2)$$

② $$\sum_{i=1}^{n} i = O(n^2)$$

$$\sum_{i=1}^{n} i^2 = 1^2+2^2+\ldots+n^2 = \frac{n(n+1)(2n+1)}{6} \in O(n^3)$$

③ $$\sum_{i=1}^{n} i^2 = O(n^3)$$

④ $$\sum_{i=1}^{n} i^d = O(n^{d+1})$$

2) $O(n^2)$ Algorithm for Maximum Subarray:

```
Alg. MS2 (A[1..n])
    mSum ← A[1]
    mRange ← (1, 1)
    for i ← 1 to n do                    → (1)
        Sum ← 0
        for j ← i to n do                → (2)
            Sum ← Sum + A[j]
            if Sum > mSum then
                mSum ← Sum
                mRange ← (i, j)
            end if
        end for
    end for
    return mRange & mSum
end MS2
```

Divide & Conquer Approach

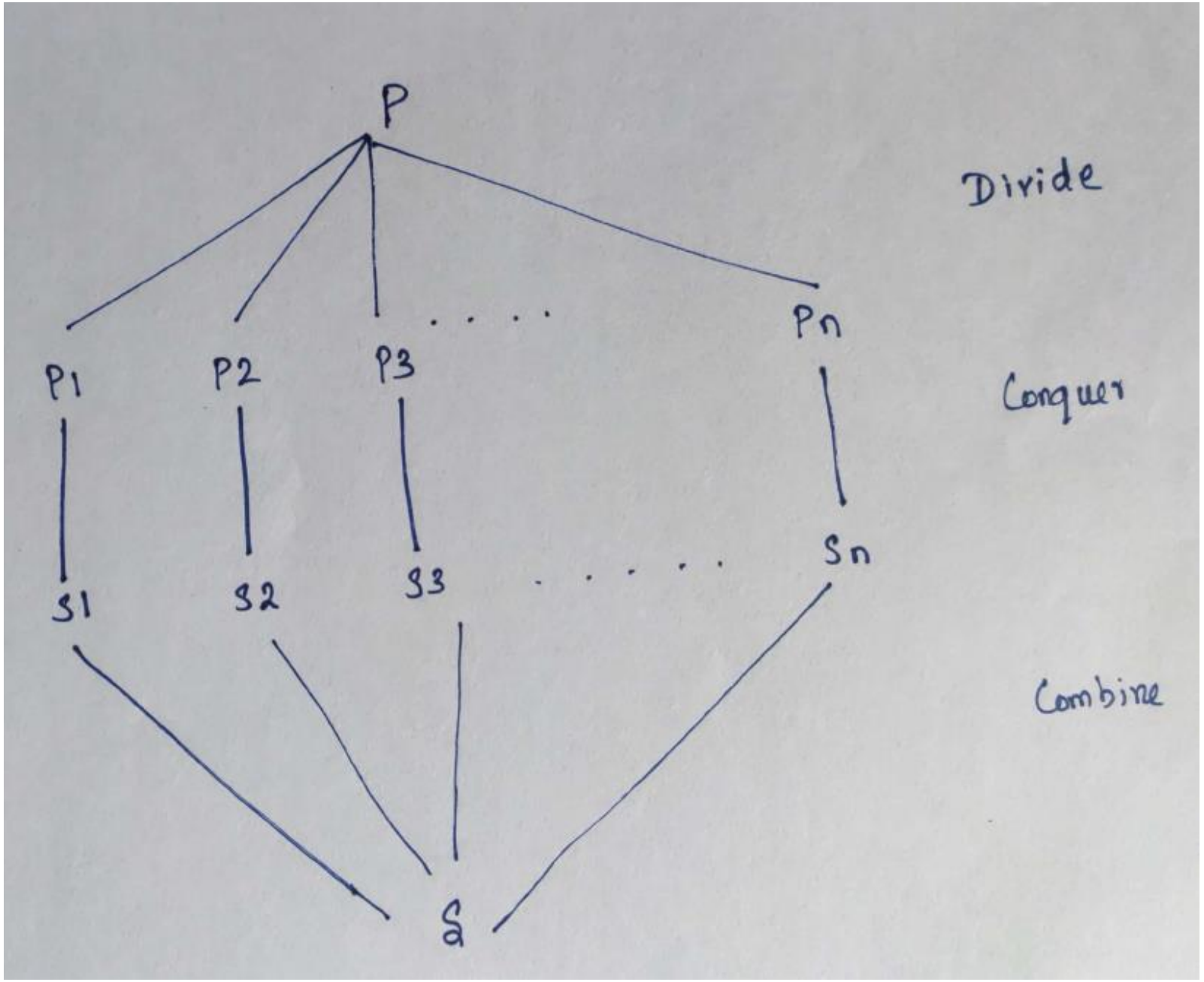
Divide & Conquer Approach: (An algorithm design technique)

3 Steps:

1. Divide : The Problem is divided into the number of Sub problems that are Smaller Instances of the Same problem

2. Conquer : The Subproblems are Solved either Recursively or Iteratively

3. Combine : The Solutions of Sub problems are Combined into the Solution to the main problem.

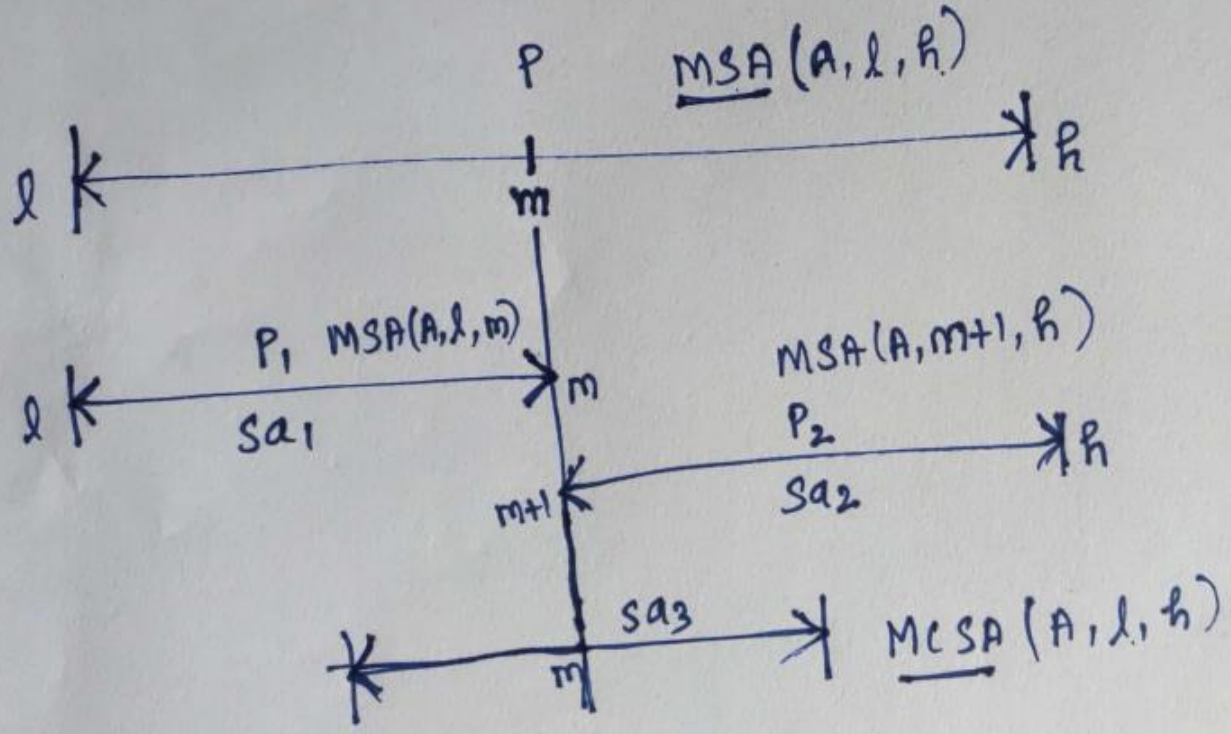

P
Divide
Pn
P1
P2
P3
Conquer
Sn
S1
S2
S3
Combine
S

3) Divide & Conquer Approach for Max. SubArray Problem:

Input : A[1..n], l, h

Output : (s, e, sum)

A	-1	3	-2	5
	1	2	3	4
	l			h



MSA(A, l, h)

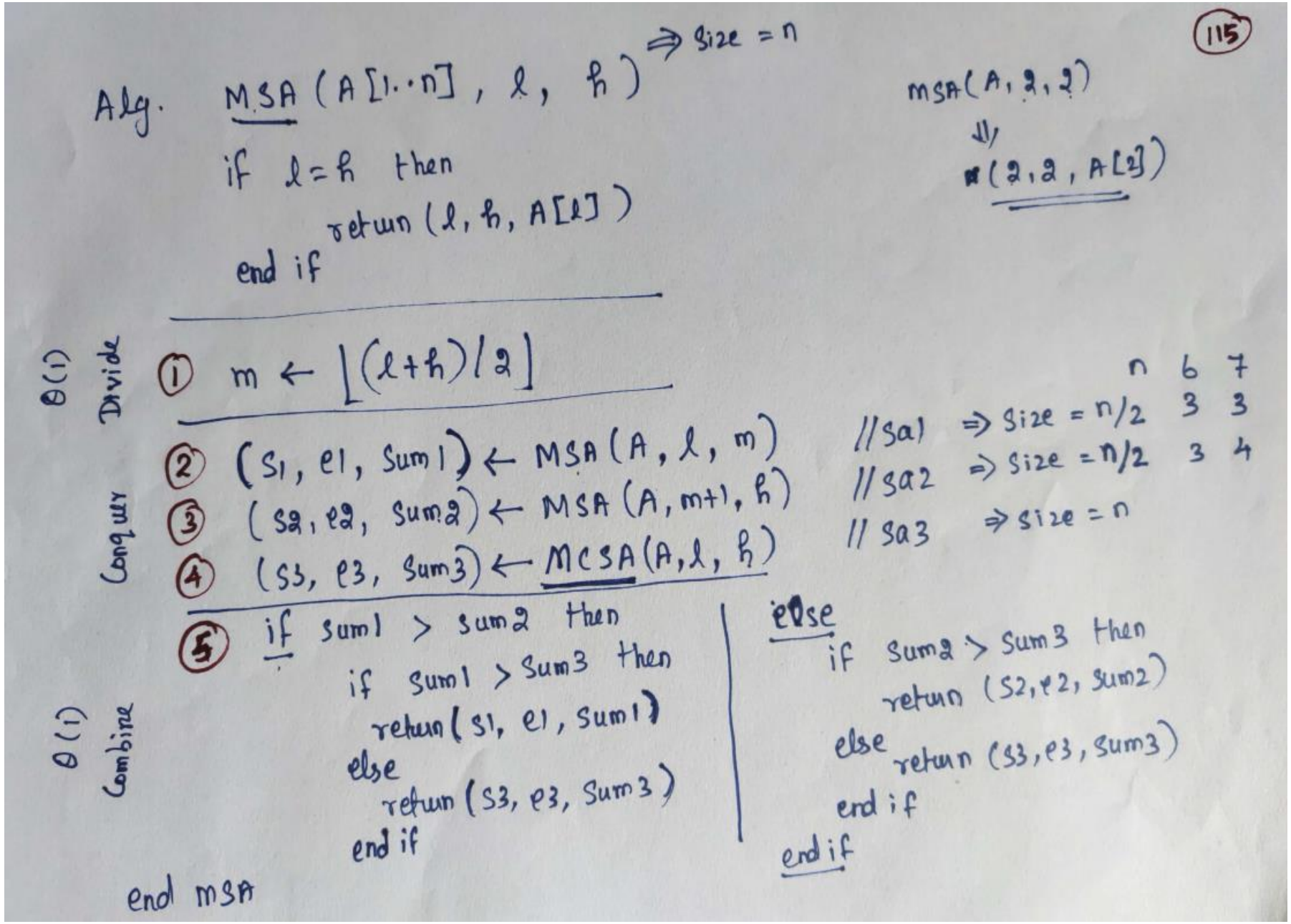
↓

MSA (A, 1, 4) ⇒ I/P

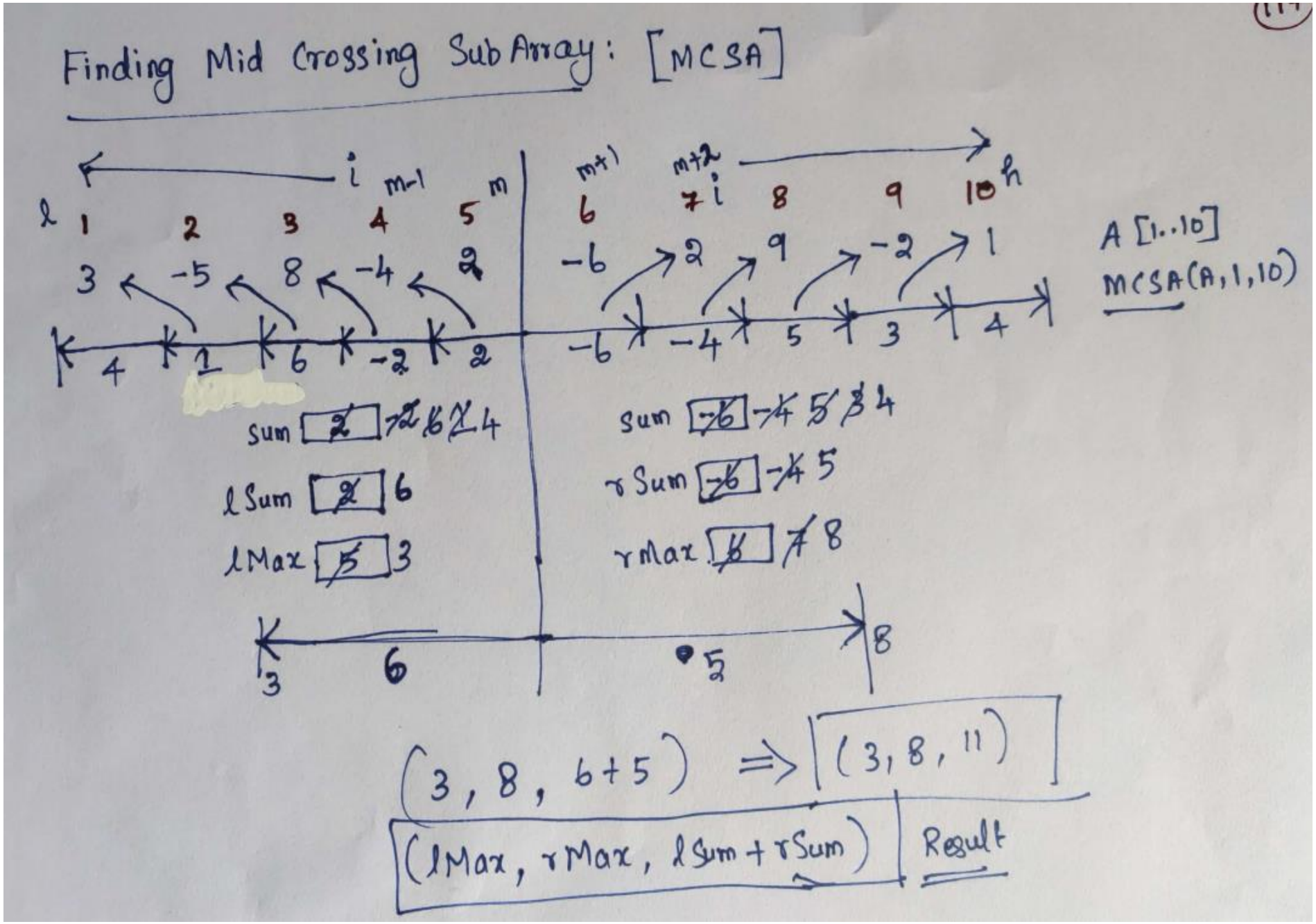
⇓

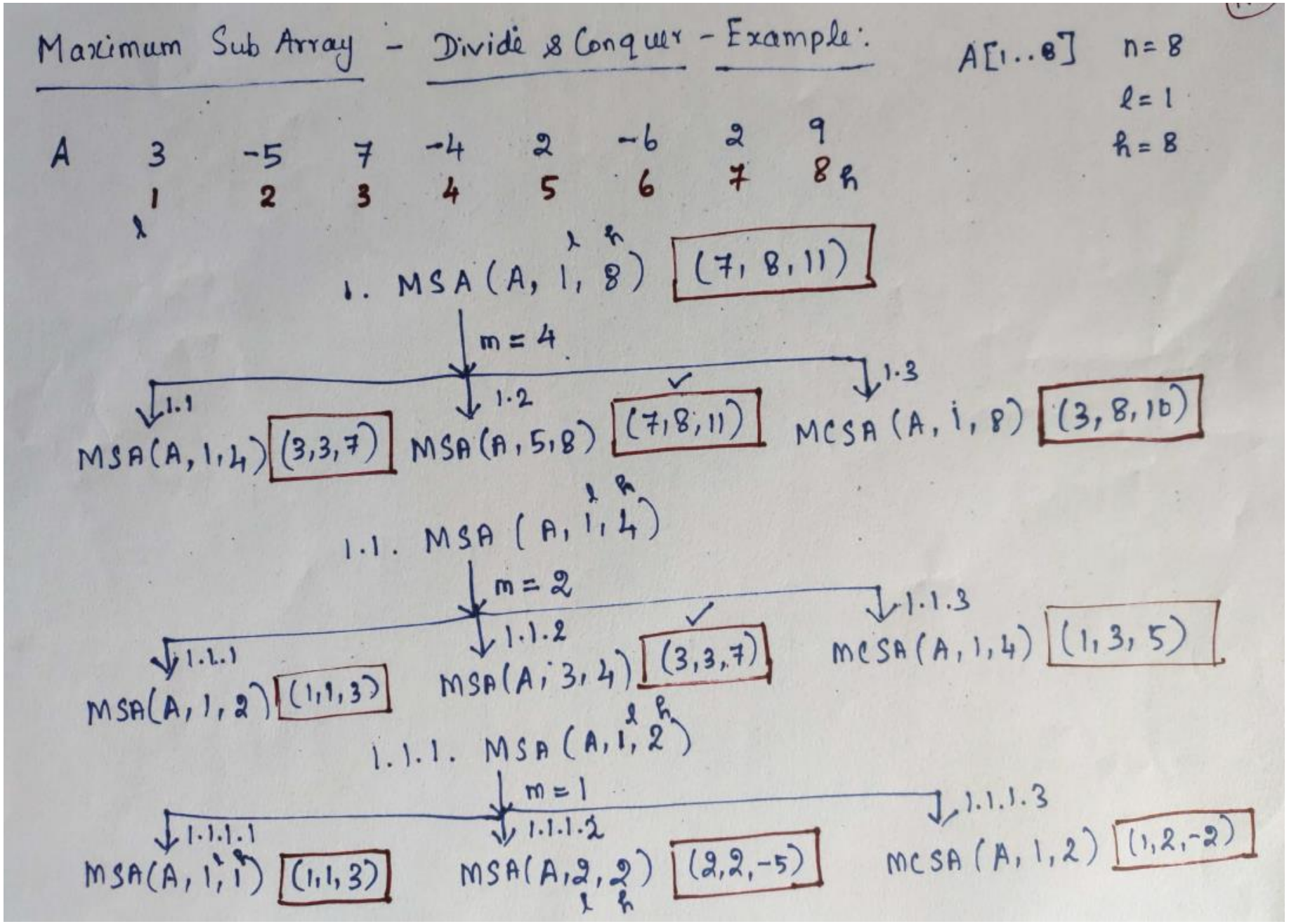
(2, 4, 6) ⇒ O/P

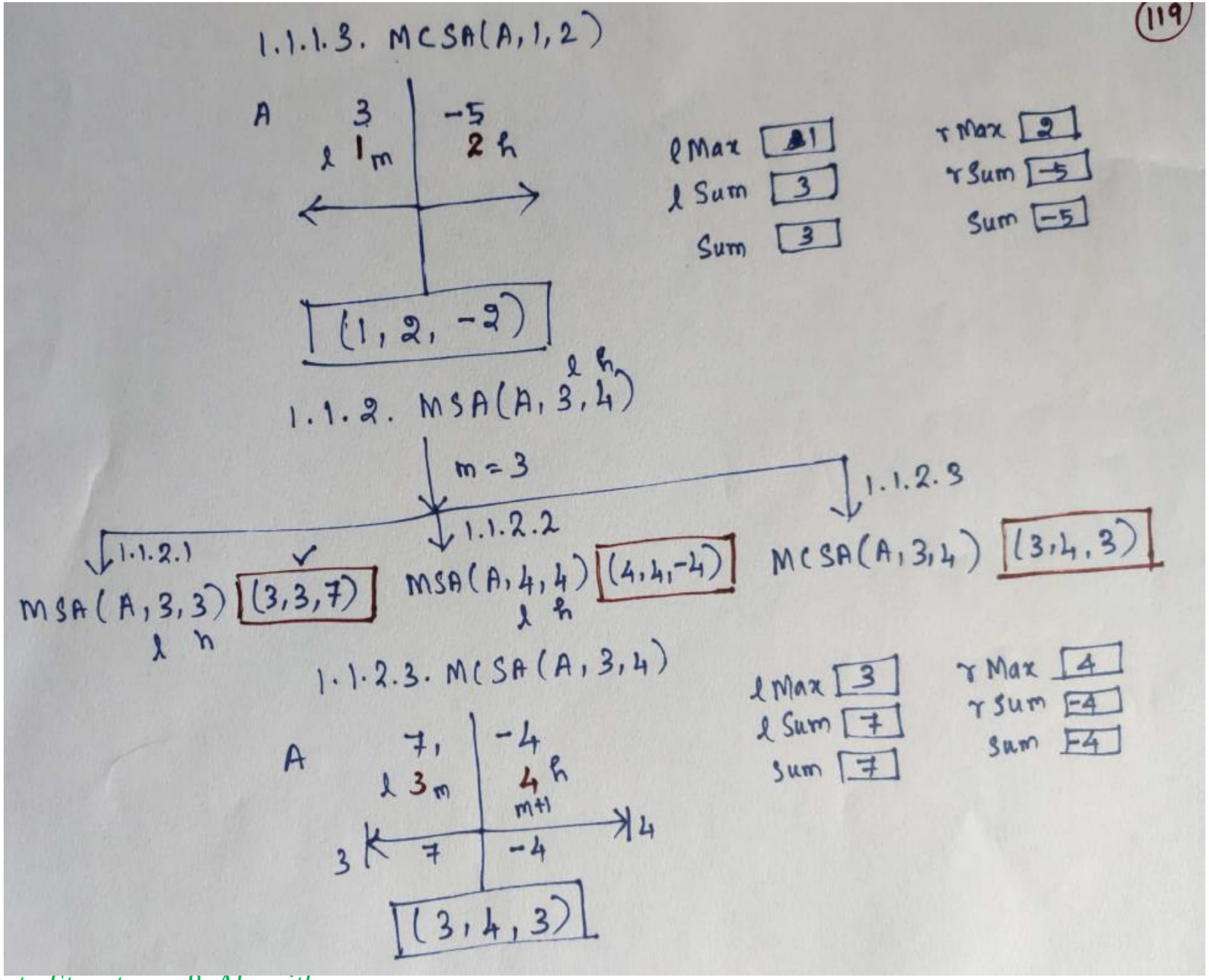
max (SA1, SA2, SA3) ⇒ Result

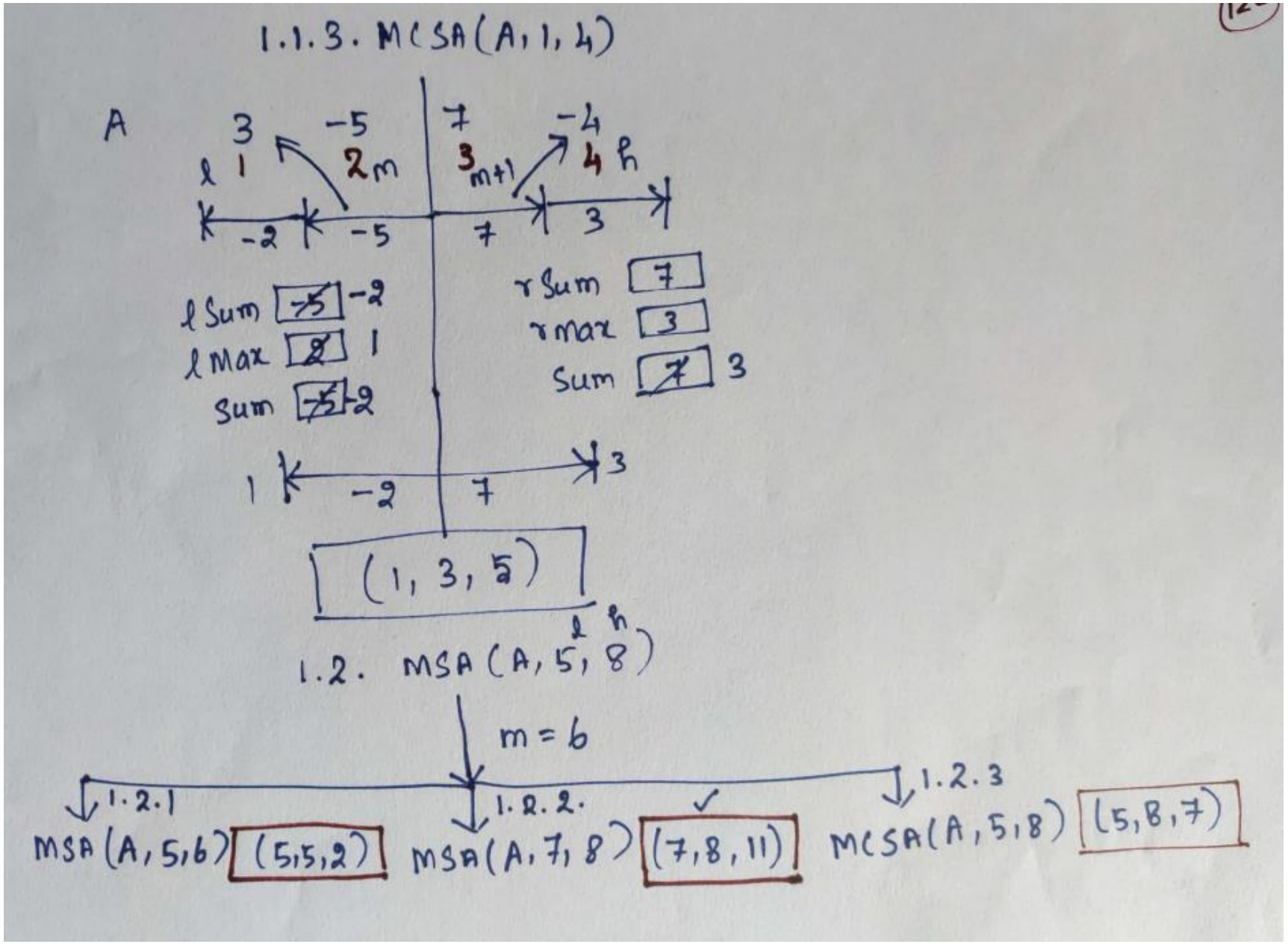

115
Alg. MSA (A[1..n], l, h) ⇒ Size = n
if l = h then
return (l, h, A[l])
end if
θ(1) Divide
① m ← ⌊(l+h)/2⌋
Conquer
② (s1, e1, sum1) ← MSA (A, l, m)
③ (s2, e2, sum2) ← MSA (A, m+1, h)
④ (s3, e3, sum3) ← MCSA (A, l, h)
θ(1) Combine
⑤ if sum1 > sum2 then
if sum1 > sum3 then
return (s1, e1, sum1)
else
return (s3, e3, sum3)
end if
else
if sum2 > sum3 then
return (s2, e2, sum2)
else return (s3, e3, sum3)
end if
end if
end MSA
MSA(A, 2, 2)
⇓
(2, 2, A[2])
// sa1 ⇒ Size = n/2
// sa2 ⇒ Size = n/2
// sa3 ⇒ Size = n
n 6 7
3 3
3 4

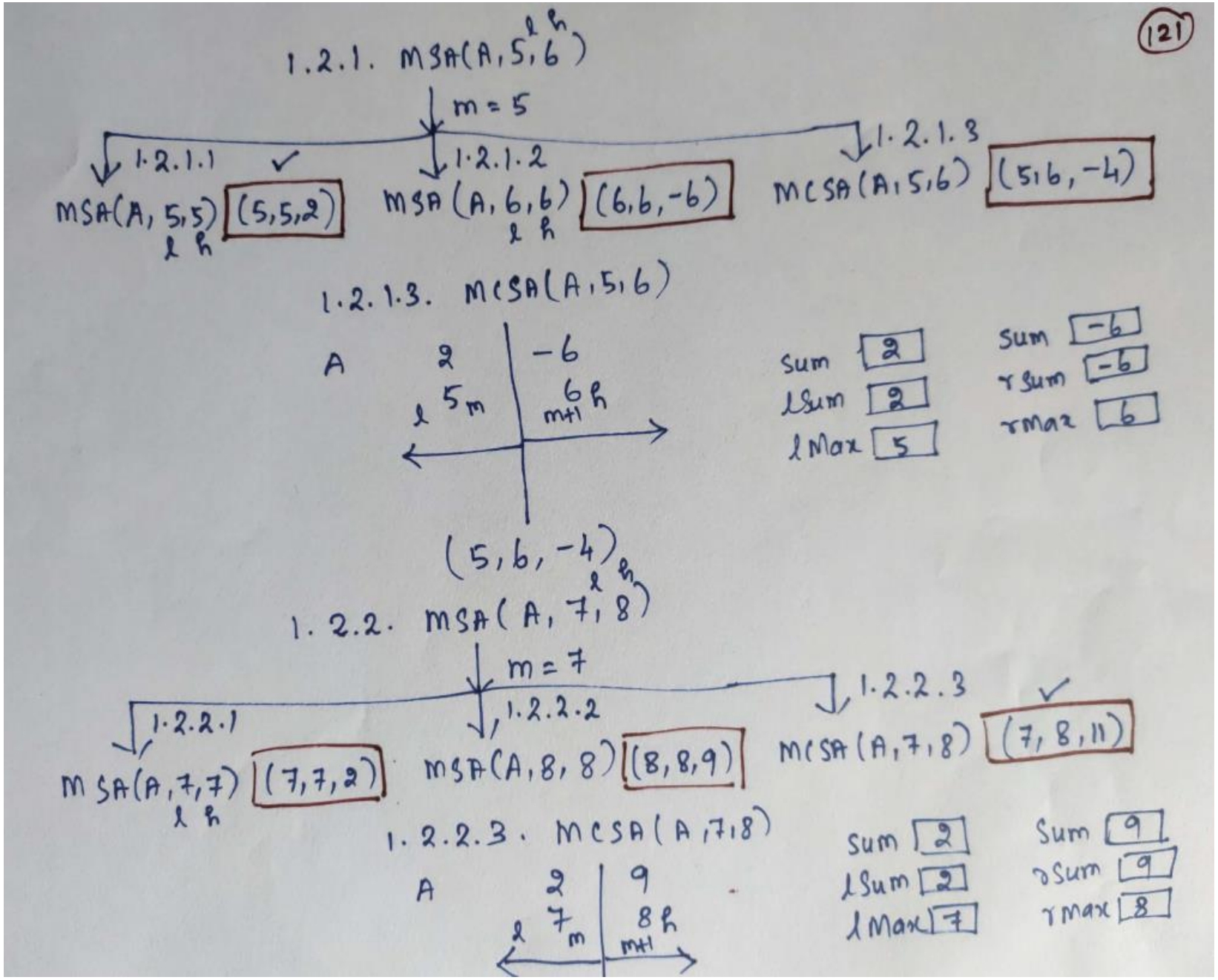
```
Alg  MCSA(A[1..n], l, h)
     m ← ⌊(l+h)/2⌋

     Sum  ← A[m]
     lSum ← A[m]
     lMax ← m

(1)  for i ← m-1 to l downwards
         Sum ← Sum + A[i]
         if Sum > lSum then
             lSum ← Sum
             lMax ← i
         end if
     end for

     Sum  ← A[m+1]
     rSum ← A[m+1]
     rMax ← m+1
(2)  for i ← m+2 to h do
         Sum ← Sum + A[i]
         if Sum > rSum then
             rSum ← Sum
             rMax ← i
         end if
     end for
     return (lMax, rMax, lSum+rSum)

end  MCSA
```

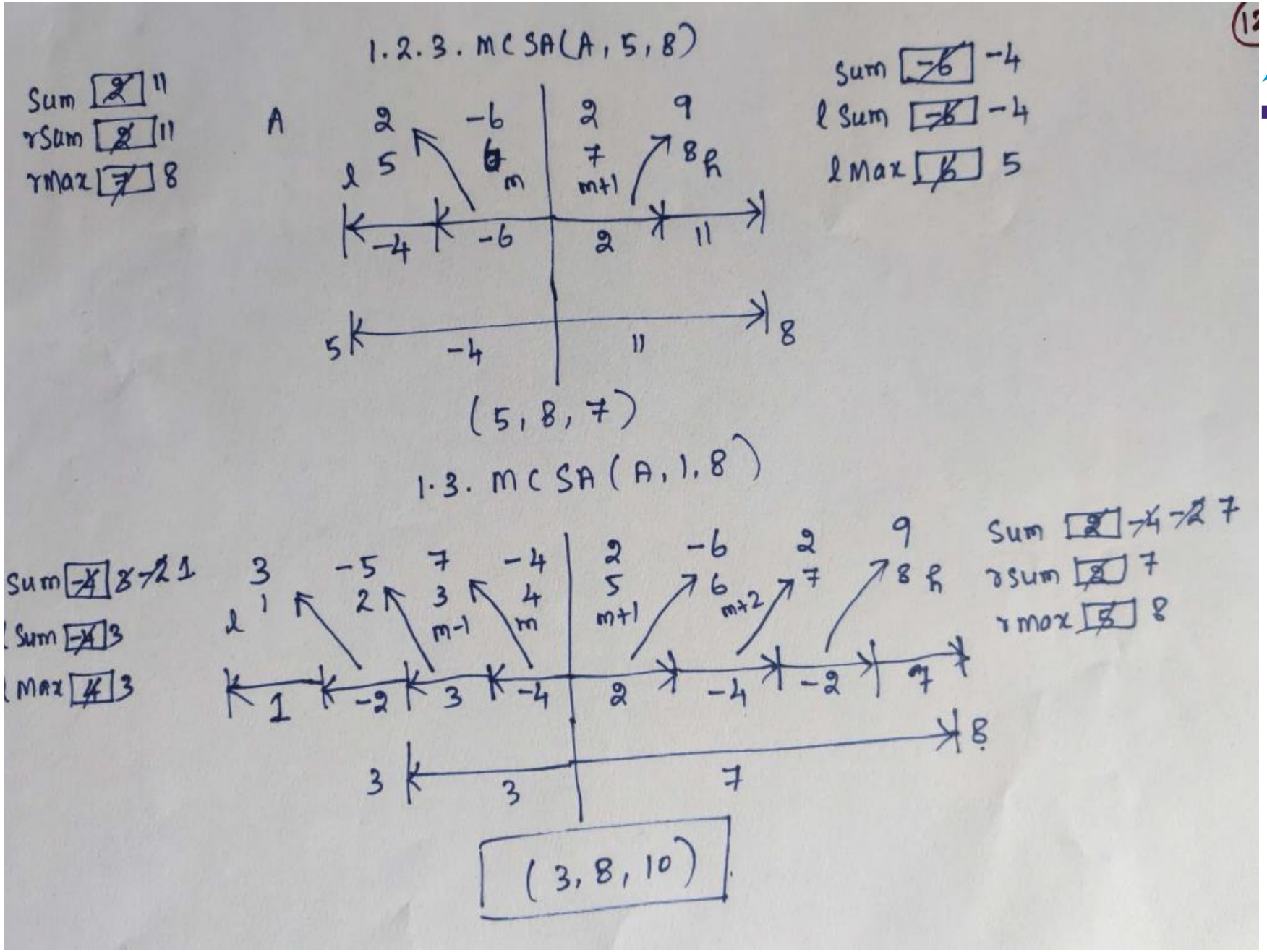

Finding Mid Crossing Sub Array: [MCSA]
A [1..10]
MCSA(A,1,10)
sum
lSum
lMax
rSum
rMax
(3, 8, 6+5) ⇒ (3, 8, 11)
(lMax, rMax, lSum + rSum)
Result


Maximum Sub Array - Divide & Conquer - Example:
A[1..8] n=8 l=1 h=8
A 3 -5 7 -4 2 -6 2 9
1 2 3 4 5 6 7 8
1. MSA(A, 1, 8) (7, 8, 11)
m = 4
1.1 MSA(A, 1, 4) (3,3,7)
1.2 MSA(A, 5, 8) (7,8,11)
1.3 MCSA(A, i, 8) (3, 8, 10)
1.1. MSA(A, 1, 4)
m = 2
1.1.1 MSA(A, 1, 2) (1,1,3)
1.1.2 MSA(A, 3, 4) (3,3,7)
1.1.3 MCSA(A, 1, 4) (1, 3, 5)
1.1.1. MSA(A, 1, 2)
m = 1
1.1.1.1 MSA(A, 1, 1) (1,1,3)
1.1.1.2 MSA(A, 2, 2) (2,2,-5)
1.1.1.3 MCSA(A, 1, 2) (1,2,-2)


119
1.1.1.3. MCSA(A,1,2)
A
3
-5
l
m
2
h
lMax 1
lSum 3
Sum 3
rMax 2
rSum -5
Sum -5
(1, 2, -2)
1.1.2. MSA(A, 3, 4)
l h
m = 3
1.1.2.1
1.1.2.2
1.1.2.3
MSA(A, 3, 3)
l h
(3,3,7)
MSA(A, 4, 4)
l h
(4,4,-4)
MCSA(A,3,4)
(3,4,3)
1.1.2.3. MCSA(A, 3, 4)
A
7,
-4
l
3
m
4
h
m+1
3
7
-4
4
lMax 3
lSum 7
Sum 7
rMax 4
rSum -4
Sum -4
(3, 4, 3)


1.1.3. MCSA(A, 1, 4)
A
3
-5
7
-4
l 1
2m
3 m+1
4 h
-2
-5
7
3
lSum -5 -2
lMax 2 1
Sum -5 -2
rSum 7
rmax 3
Sum 7 3
1
-2
7
3
(1, 3, 5)
l
h
1.2. MSA(A, 5, 8)
m = 6
1.2.1
1.2.2.
1.2.3
MSA(A, 5, 6)
(5, 5, 2)
MSA(A, 7, 8)
(7, 8, 11)
MCSA(A, 5, 8)
(5, 8, 7)


121
1.2.1. MSA(A,5,6)
l h
m = 5
1.2.1.1
MSA(A,5,5)
l h
(5,5,2)
1.2.1.2
MSA(A,6,6)
l h
(6,6,-6)
1.2.1.3
MCSA(A,5,6)
(5,6,-4)
1.2.1.3. MCSA(A,5,6)
A
2
-6
l
5 m
m+1
6 h
Sum 2
lSum 2
lMax 5
Sum -6
rSum -6
rMax 6
(5,6,-4)
1.2.2. MSA(A,7,8)
l h
m = 7
1.2.2.1
MSA(A,7,7)
l h
(7,7,2)
1.2.2.2
MSA(A,8,8)
(8,8,9)
1.2.2.3
MCSA(A,7,8)
(7,8,11)
1.2.2.3. MCSA(A,7,8)
A
2
9
l
7 m
m+1
8 h
Sum 2
lSum 2
lMax 7
Sum 9
rSum 9
rMax 8


1.2.3. MCSA(A, 5, 8)
Sum 2 11
rSum 2 11
rMax 7 8
A
2 -6 2 9
l 5 6 7 8 h
m m+1
-4 -6 2 11
5 -4 11 8
Sum -6 -4
lSum -6 -4
lMax 6 5
(5, 8, 7)
1.3. MCSA(A, 1, 8)
Sum -4 8 -2 1
lSum -4 3
lMax 4 3
3 -5 7 -4 2 -6 2 9
l 1 2 3 4 5 6 7 8 h
m-1 m m+1 m+2
1 -2 3 -4 2 -4 -2 7
3 3 7 8
Sum 2 -4 -2 7
rSum 2 7
rMax 5 8
(3, 8, 10)

Analysis of Divide & Conquer Algorithm for Maximum Sub Array Problem:

- 1. MSA
- 2. MCSA

Analyze the Complexity of MCSA: (for size n)

Loop ① $\Rightarrow \sum_{i=l}^{m-1} 1 = m-1-l+1 \Rightarrow m-l$

Loop ② $\Rightarrow \sum_{i=m+2}^{h} 1 = h-(m+2)+1 \Rightarrow h-m-1$

Time Complexity on 'n':

$T(n) = m-l+h-m-1$

$= h-l-1$

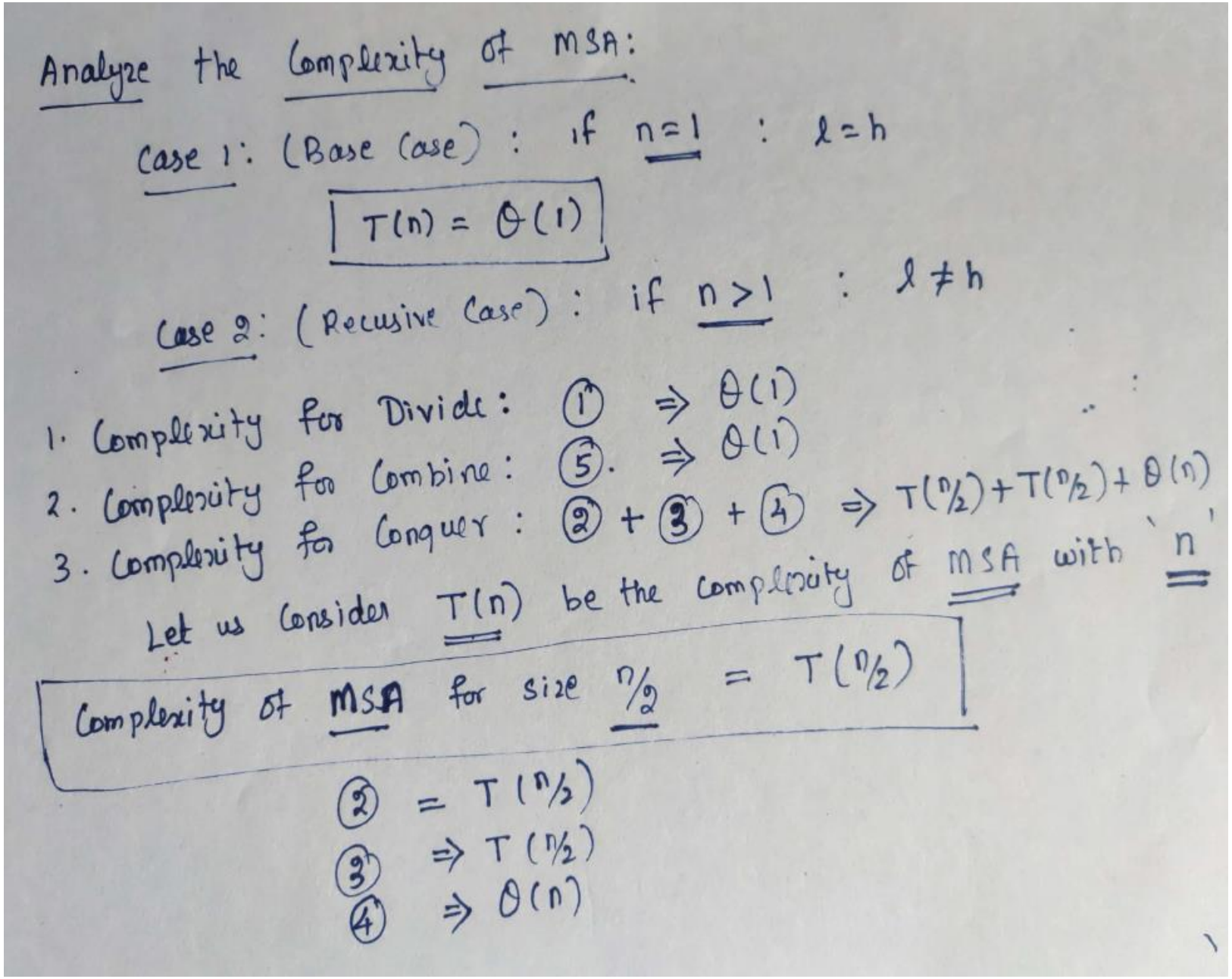
$= n-1-1 \Rightarrow n-2 \in O(n)$

For 'n' input:

$l = 1$

$h = n$

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


Analyze the Complexity of MSA:
Case 1: (Base case) : if n=1 : l=h
T(n) = Θ(1)
Case 2: (Recusive Case) : if n>1 : l≠h
1. Complexity for Divide: (1) ⇒ Θ(1)
2. Complexity for Combine: (5) ⇒ Θ(1)
3. Complexity for Conquer: (2) + (3) + (4) ⇒ T(n/2)+T(n/2)+Θ(n)
Let us Consider T(n) be the Complexity of MSA with n
Complexity of MSA for size n/2 = T(n/2)
(2) = T(n/2)
(3) ⇒ T(n/2)
(4) ⇒ Θ(n)

$T(n) = ① + ② + ③ + ④ + ⑤$

$= \underline{\Theta(1)} + T(n/2) + T(n/2) + \underline{\Theta(n)} + \underline{\Theta(1)}$

$$\boxed{T(n) = 2 \cdot T(n/2) + \Theta(n) \quad \text{if } n>1}$$

$$T(n) = \begin{cases} 2 \cdot T(n/2) + \Theta(n) & , \text{if } n>1 \\ \Theta(1) & , \text{if } n=1 \end{cases}$$

⇒ Rewrence - Need to be solved

1. Recursion Tree
2. Substitution
3. Master Theorem

$T(n) = 2 \cdot T(n/2) + \Theta(n)$

$a = 2 \quad b = 2 \quad k = 1 \quad P = 0$

$a \quad b^k \Rightarrow 2 \quad 2^1 \Rightarrow \boxed{2 = 2} \Rightarrow \boxed{a = b^k} \Rightarrow$ Case 2

$P = 0 \Rightarrow$ Subcase (iii) $\Rightarrow$ Solution: $T(n) = \Theta(n^{\log_b a} \cdot \log^{(k+1)} n) = \Theta(n^{\log_2 2} \cdot \log^1 n)$

$(P < -1)$

Complexity MSA for size 'n' $\boxed{T(n) = \Theta(n \cdot \log n)}$

Method for Solving Recurrence:

$$T(n) = a \cdot T(n/b) + \theta(n^k \cdot \log^p n)$$

Case 1: $a > b^k$ $\Rightarrow$ Solution: $T(n) = \theta(n^{\log_b a})$

Case 2: $a = b^k$

i) $P < -1$ $\Rightarrow$ Solution: $T(n) = \theta(n^{\log_b a})$

ii) $P = -1$ $\Rightarrow$ Solution: $T(n) = \theta(n^{\log_b a} \cdot \log^2 n)$

iii) $P > -1$ $\Rightarrow$ Solution: $T(n) = \theta(n^{\log_b a} \cdot \log^{P+1} n)$

Case 3: $a < b^k$

i) $P < 0$ $\Rightarrow$ Solution: $T(n) = \theta(n^k)$

ii) $P \geq 0$ $\Rightarrow$ Solution: $T(n) = \theta(n^k \cdot \log^P n)$

Divide & Conquer Approach

Merge Sort – Algorithm, Example & Analysis

Merge Sort: (Divide & Conquer Approach)

- MergeSort (A[0..n-1], p, r)
- Merge (A[0..n-1], p, q, r)

p → Start Index
r → End "
q → Middle Index.

Merge:

n = 8

p q r

⇒ Merge (A, 0, 3, 7)

⇓

Sort the elements from 0 to 7

A	19	28	45	67	12	26	31	39
	0	1	2	3	4	5	6	7
	p			q	q+1			r

$n_1 = q - p + 1$
$= 3 - 0 + 1 \Rightarrow \boxed{n_1 = 4}$

$n_2 = r - (q+1) + 1$
$= r - q \Rightarrow \boxed{n_2 = 4}$
$= 7 - 3$

[0..n_1] L

19	28	45	67	∝
0	1	2	3	4
i				

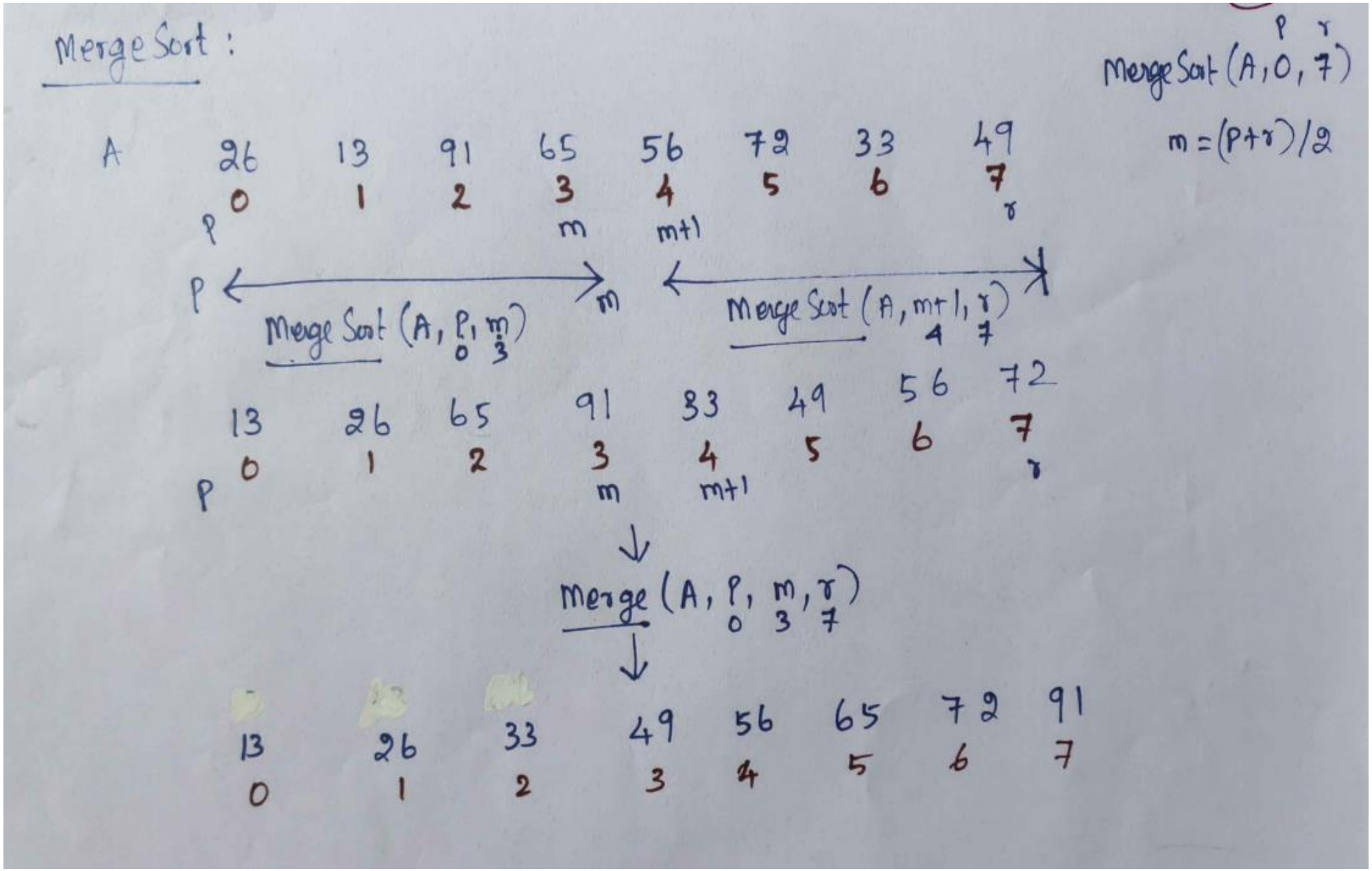
R [0..n_2]

12	26	31	39	∝
0	1	2	3	4
j				

K → p to r

A

12	19	26	28	31	39	45	67	8
0	1	2	3	4	5	6	7	k


Merge Sort :
p r
Merge Sort (A, 0, 7)
m = (p+r)/2
A
26 13 91 65 56 72 33 49
0 1 2 3 4 5 6 7
p m m+1 r
p m
Merge Sort (A, p, m)
0 3
Merge Sort (A, m+1, r)
4 7
13 26 65 91 33 49 56 72
0 1 2 3 4 5 6 7
p m m+1 r
Merge (A, p, m, r)
0 3 7
13 26 33 49 56 65 72 91
0 1 2 3 4 5 6 7

```
Alg. Merge (A[0..n-1], P, q, r)
    n1 ← q-P+1
    n2 ← r-q
    //Let 2 Arrays L[0..n1] and R[0..n2]
     i ← 0
     j ← 0
①   for K ← p to r do
        if K <= q then
            L[i] ← A[K]
            i ← i+1
        else
            R[j] ← A[K]
            j ← j+1
        end if
    end for
    L[n1] ← R[n2] ← ∝
```

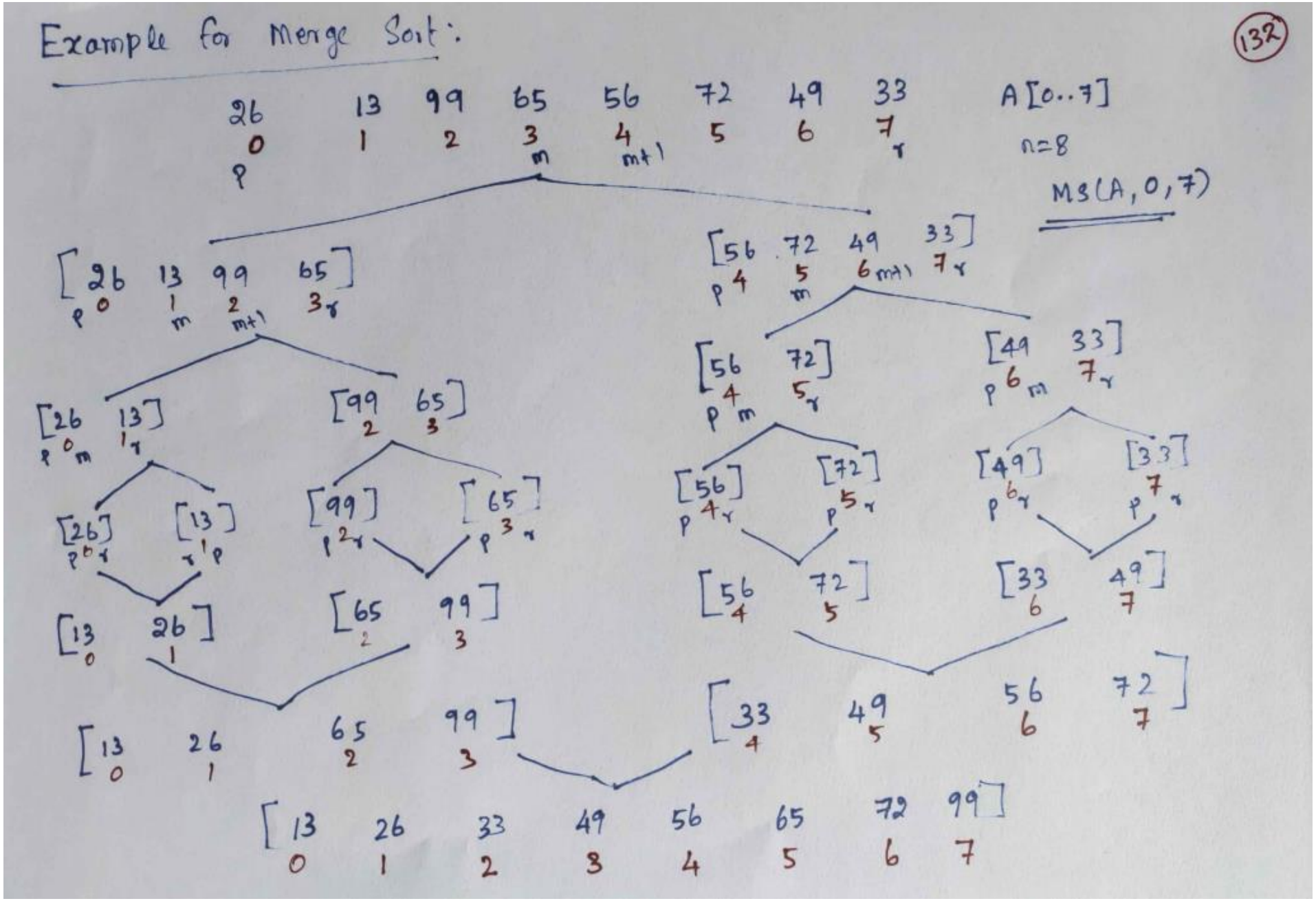
```
    i ← 0
    j ← 0
②   for k ← P to r do
        if L[i] < R[j] then
            A[K] ← L[i]
            i ← i+1
        else
            A[K] ← R[j]
            j ← j+1
        end if
    end for
end Merge.
```

```
Alg MergeSort (A[0..n-1], p, r) // => if 'n' is size
    if p >= r then
Base    return;
    end if
```

Rewsive

```
(1) m ← ⌊(p+r)/2⌋                          → Θ(1)
(2) MergeSort (A, p, m)        // size = n/2   → T(n/2)
(3) MergeSort (A, m+1, r)      // size = n/2   → T(n/2)
(4) Merge (A, p, m, r)                      → Θ(n)

end MergeSort
```

Example for Merge Sort:



Analysis of Merge Sort:

- Analyze Merge
- Analyze MergeSort

Merge:

Loop (1) $\Rightarrow \sum_{k=p}^{r} 1 = r - p + 1$

Loop (2) $\Rightarrow \sum_{k=p}^{r} 1 \Rightarrow r - p + 1$

$T(n)$ for Merge $=$ (1) + (2) $\Rightarrow r - p + 1 + r - p + 1$

$= 2(r - p) + 2$

$= 2(n - 1 - 0) + 2 = 2n - 2 + 2$

$T(n) = 2n \in O(n)$

$\boxed{T(n) = \Theta(n)}$

Merge (A, p, q, r)

↓ A [0..n-1]

Merge (A, 0, _, n-1)

↓ ↓

$\boxed{p = 0 \quad r = n-1}$

Merge Sort:

Base Case: $(p = r)$, $(n = 1)$

$$T(n) = \theta(1), \text{ if } n = 1$$

Recursive Case: $(n > 1)$

Let $T(n)$ be the Complexity of Merge Sort with size 'n'

The Complexity of Merge Sort with size $n/2$ $\Rightarrow T(n/2)$

$$T(n) = (1) + (2) + (3) + (4)$$

$$= \theta(1) + T(n/2) + T(n/2) + \theta(n) = 2 \cdot T(n/2) + \theta(n)$$

$$T(n) = \begin{cases} 2 \cdot T(n/2) + \theta(n), & \text{if } n > 1 \\ \theta(1), & \text{if } n = 1 \end{cases}$$

Recurrence

Solving Recurrence:

$$T(n) = 2 \cdot T(n/2) + \theta(n) \qquad \theta(n^1 \cdot \log^0 n)$$

$a = 2 \quad b = 2 \quad k = 1 \quad p = 0$

$a \quad b^k \Rightarrow 2 \quad 2^1 \Rightarrow 2 = 2 \Rightarrow \boxed{a = b^k}$

$p = 0 \Rightarrow$ Case 2 Subcase (iii)

Solution: $T(n) = \theta(n^{\log_b a} \cdot \log^{p+1} n)$

$T(n) = \theta(n^{\log_2 2} \cdot \log^{0+1} n)$

$$\boxed{T(n) = \theta(n \cdot \log n)}$$

$$\boxed{T(n) = \begin{cases} \theta(n \log n) & \text{if } n > 1 \\ \theta(1) & \text{if } n = 1 \end{cases}}$$

Complexity Analysis of Divide & Conquer Approach

Divide & Conquer Approach:

Three steps in this approach.

1. Divide

→ The problem is divided into the number of sub problems that are smaller instances of same problem.

2. Conquer:

→ In this step, the sub problems are solved either recursively or using iterations.

3. Combine:

→ The solutions of sub problems are combined into the solution for the original problem.

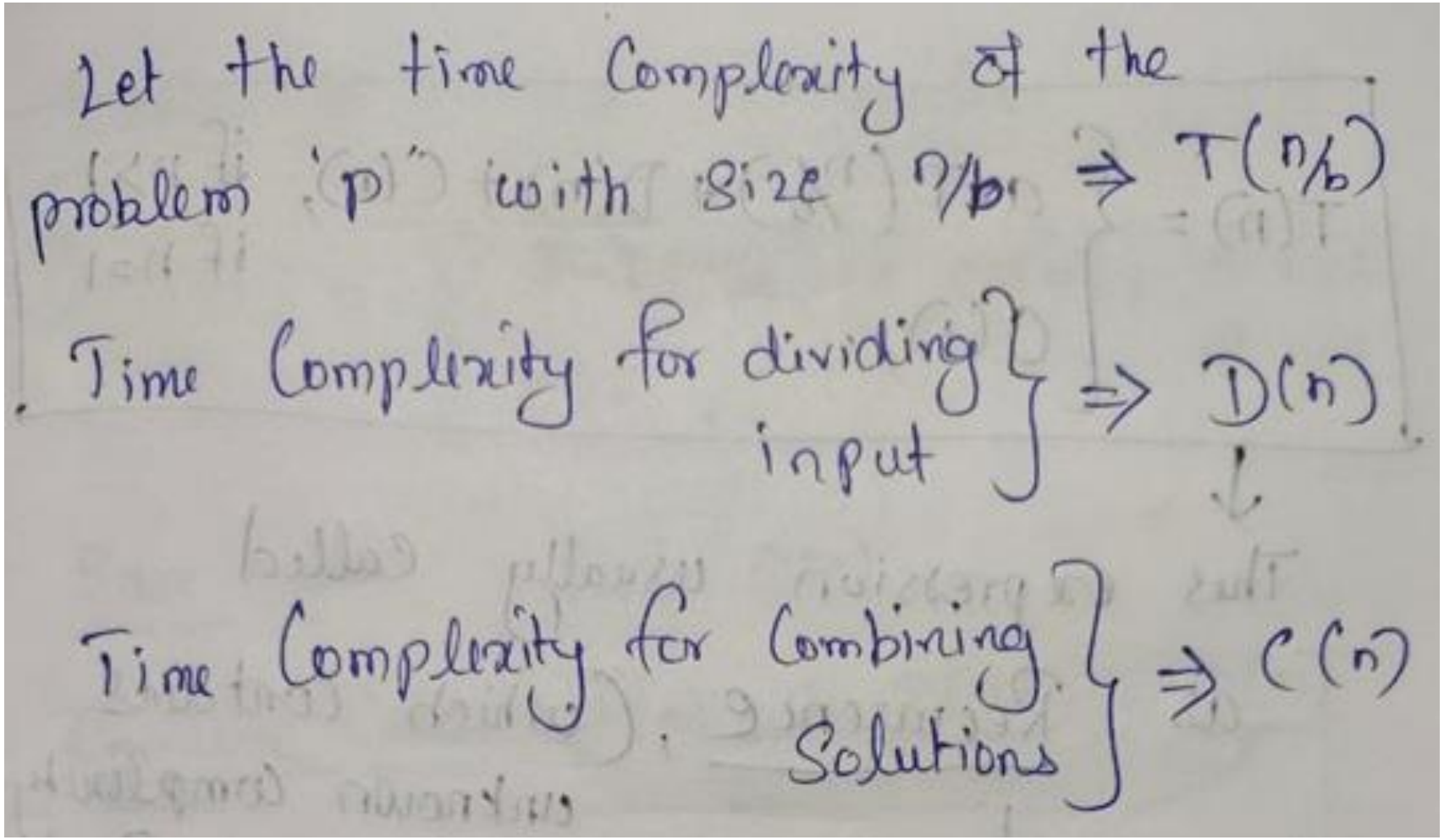
Complexity Analysis of Divide & Conquer

Let the problem 'P' of size 'n'. Suppose the problem is divided into 'a' no. of subproblems.

i.e, $P_1, P_2, \ldots P_a$.

Let the size of the subproblems is : n/b (unequally divide)

: n/a (equally divide)


Let the time Complexity of the problem 'P' with size n/b ⇒ T(n/b)
Time Complexity for dividing input ⇒ D(n)
Time Complexity for Combining Solutions ⇒ C(n)

Let the time Complexity of the problem P with size 1 ⇒ O(1)

Then the Time Complexity of the problem P with size n ⇒ T(n)

$$T(n) = \begin{cases} (\text{Sum of Complexity of all subproblems}) + (\text{Complexity for divide input}) + (\text{Complexity for Combine solutions}), & \text{if } n > 1 \\ O(1), & \text{if } n = 1 \end{cases}$$

$$T(n) = \begin{cases} aT(n/b) + D(n) + C(n), & \text{if } n \geq 1 \\ O(1), & \text{if } n = 1 \end{cases}$$

This expression usually called as Recurrence, (which contains unknown complexity in expression)

Need to be solved by using any of the following methods.

Solving Recurrences

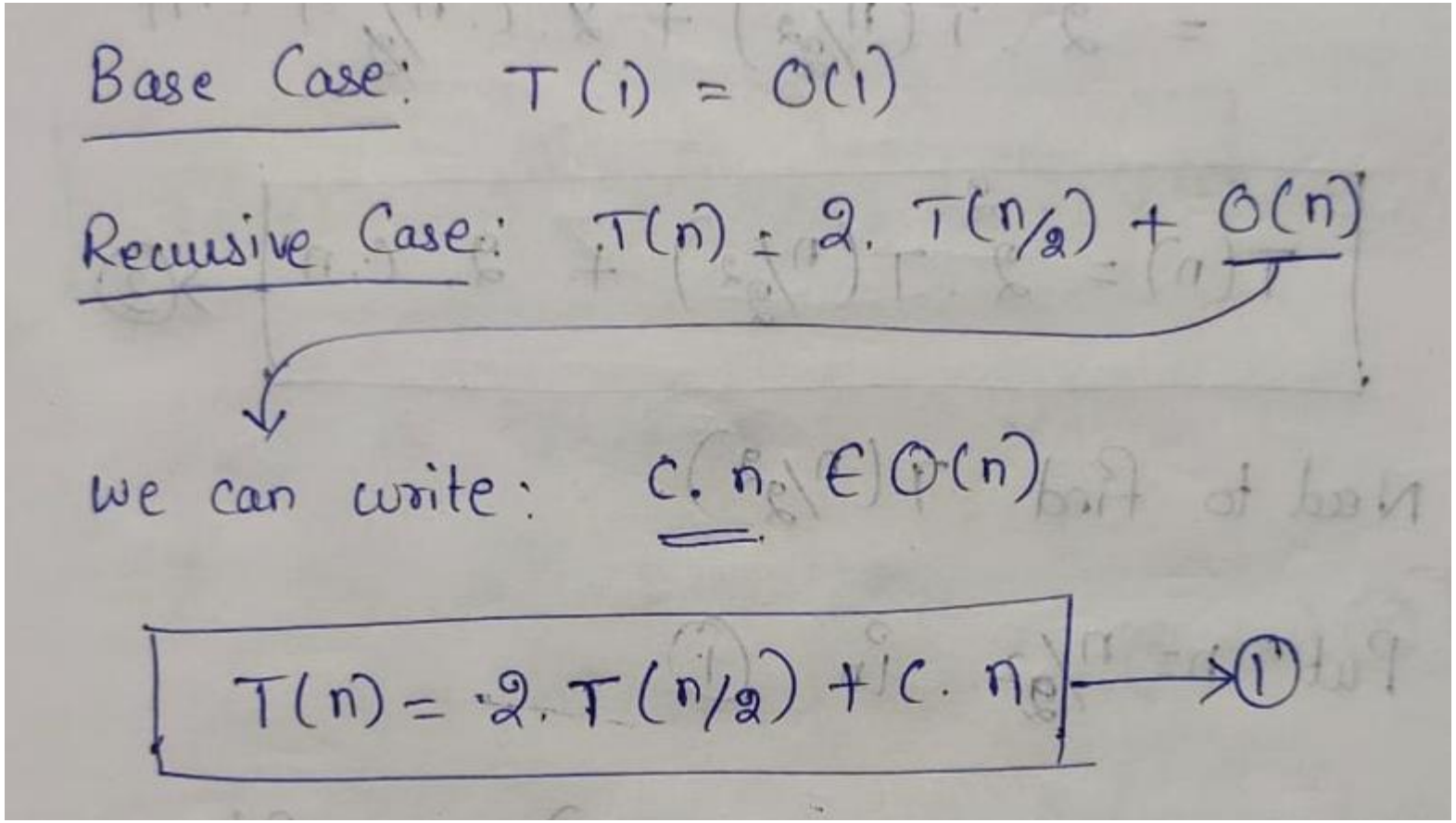
Solving Recurrences

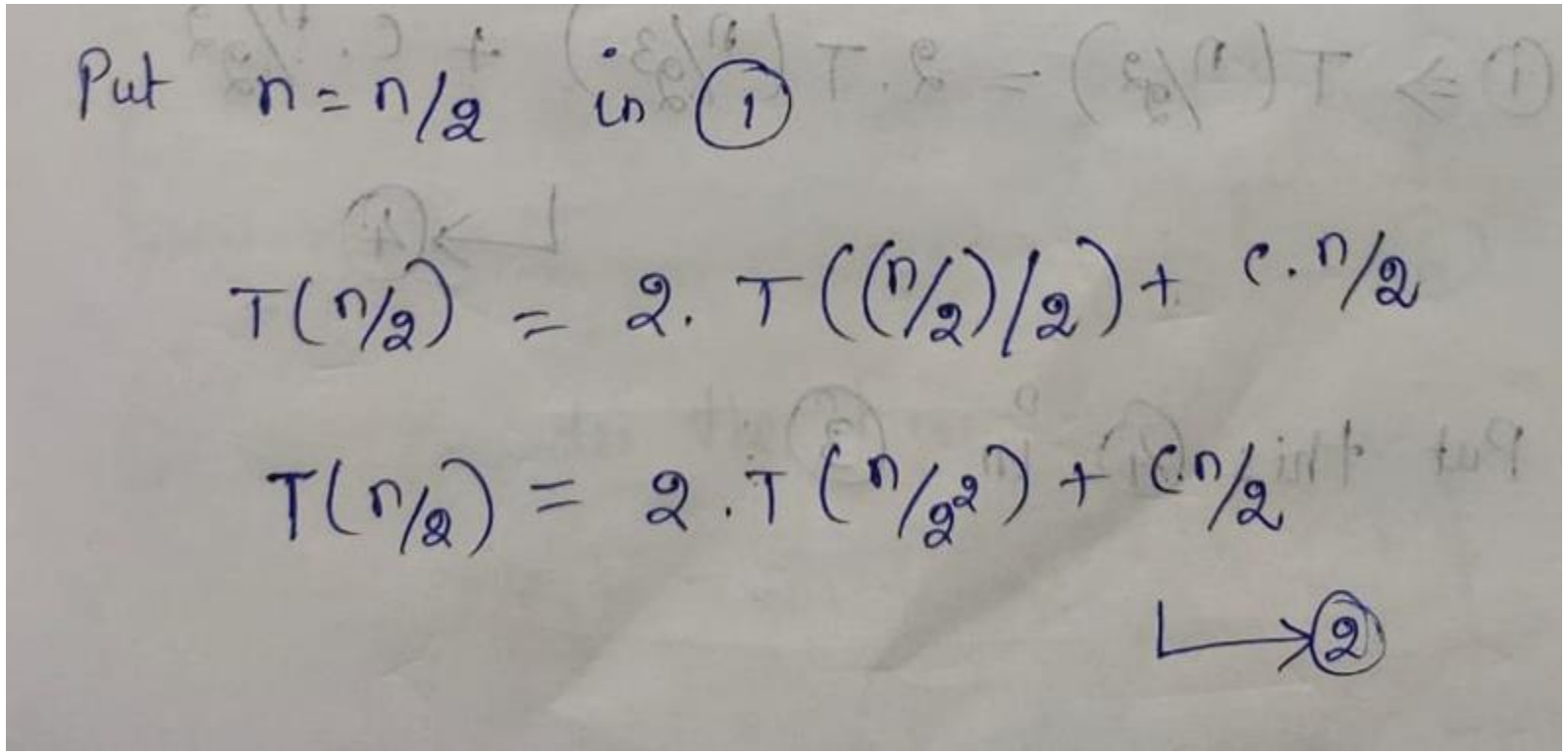
- ✓ Iterative Method
- ✓ Recursion Tree Method
- ✓ Master Theorem
- ✓ Substitution Method

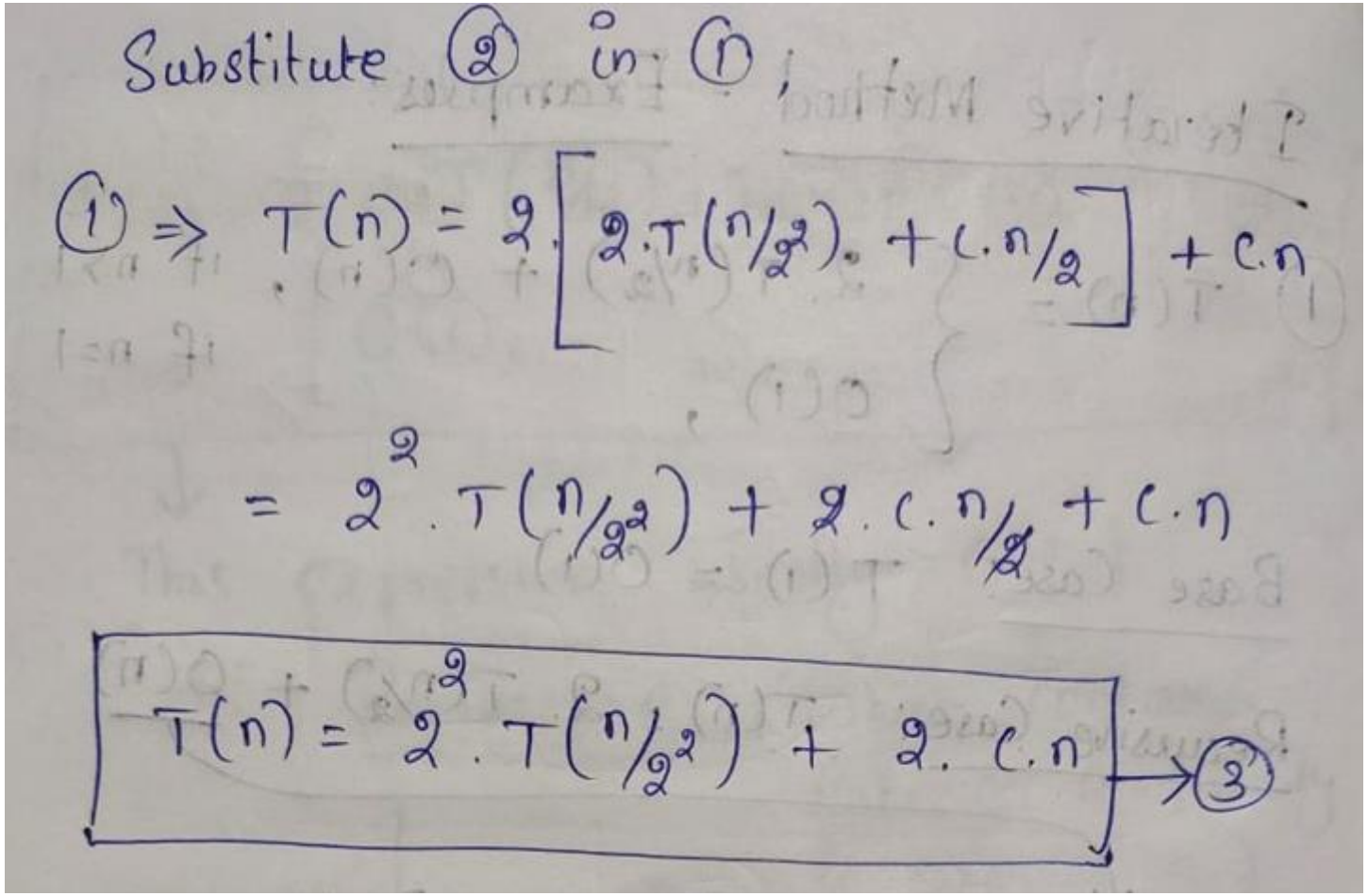
Solving Recurrences –Iterative Method

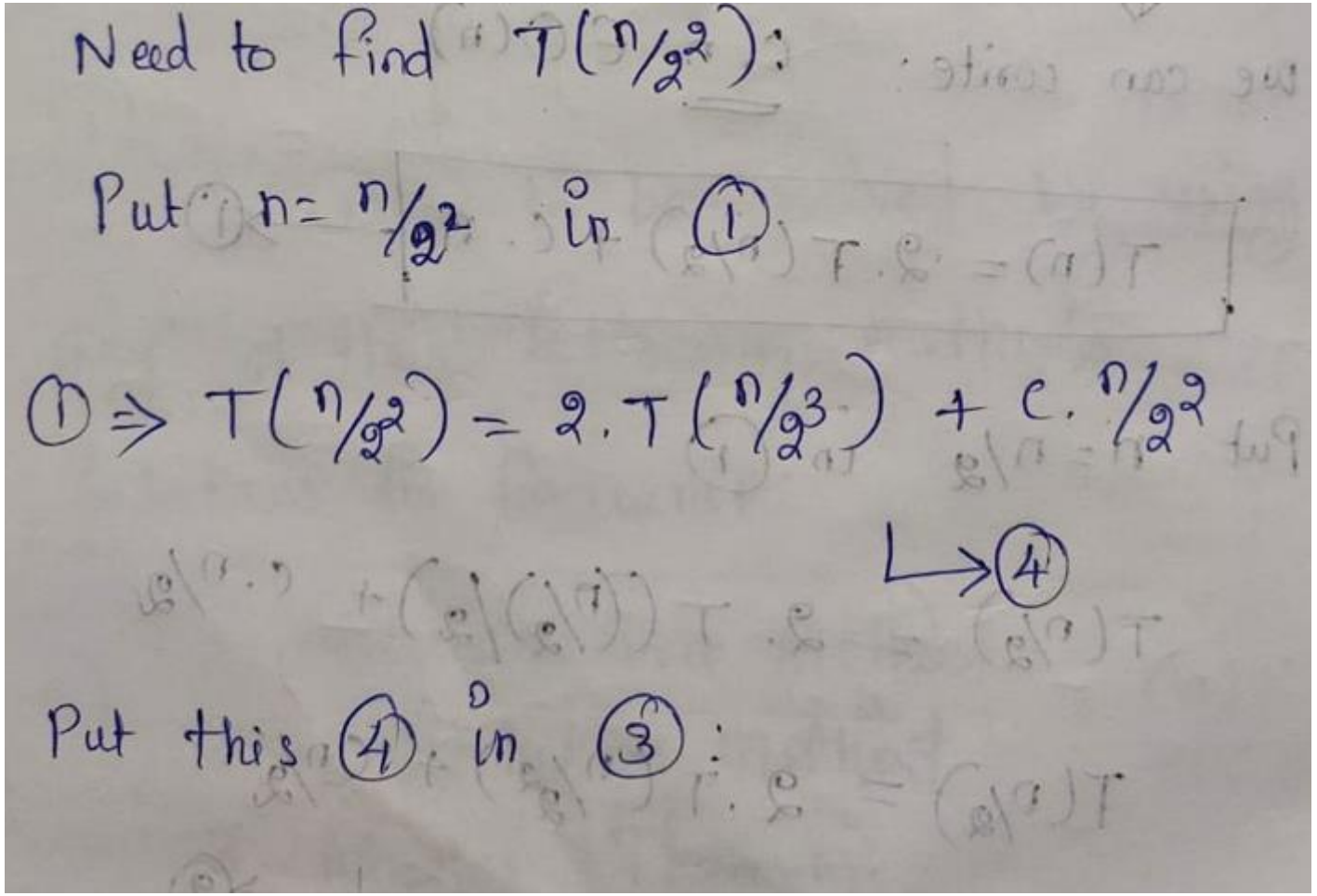
Iterative Method - Examples:

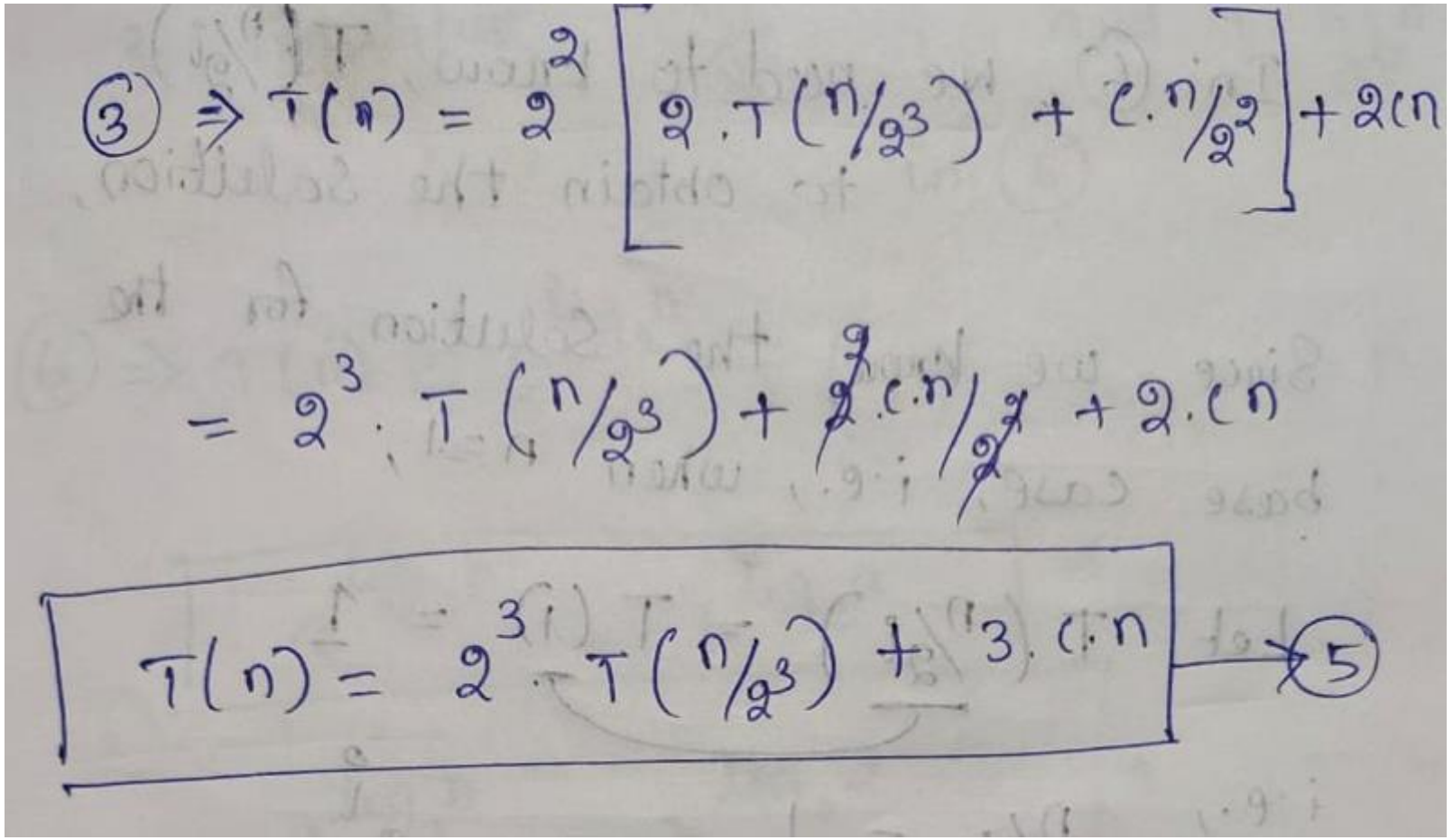
(1) $$T(n) = \begin{cases} 2 \cdot T(n/2) + \Theta(n), & \text{if } n > 1 \\ \Theta(1), & \text{if } n = 1 \end{cases}$$


Base Case: T(1) = O(1)
Recursive Case: T(n) = 2. T(n/2) + O(n)
we can write: c. n ∈ O(n)
T(n) = 2. T(n/2) + c. n
(1)


Put n = n/2 in (1)
T(n/2) = 2. T((n/2)/2) + c.n/2
T(n/2) = 2.T(n/2^2) + cn/2
(2)


Substitute ② in ①;
① ⇒ $T(n) = 2\left[2 \cdot T(n/2) + c \cdot n/2\right] + c \cdot n$
$= 2^2 \cdot T(n/2^2) + 2 \cdot c \cdot n/2 + c \cdot n$
$T(n) = 2^2 \cdot T(n/2^2) + 2 \cdot c \cdot n$ → ③


Need to find T(n/2²):
Put n = n/2² in (1)
(1) ⇒ T(n/2²) = 2.T(n/2³) + c.n/2²
→ (4)
Put this (4) in (3):


③ ⇒ T(n) = 2^2 [2 . T(n/2^3) + c . n/2^2] + 2cn
= 2^3 . T(n/2^3) + 2 . cn/2^2 + 2 . cn
T(n) = 2^3 . T(n/2^3) + 3 . cn
⑤

Keep on iterating,

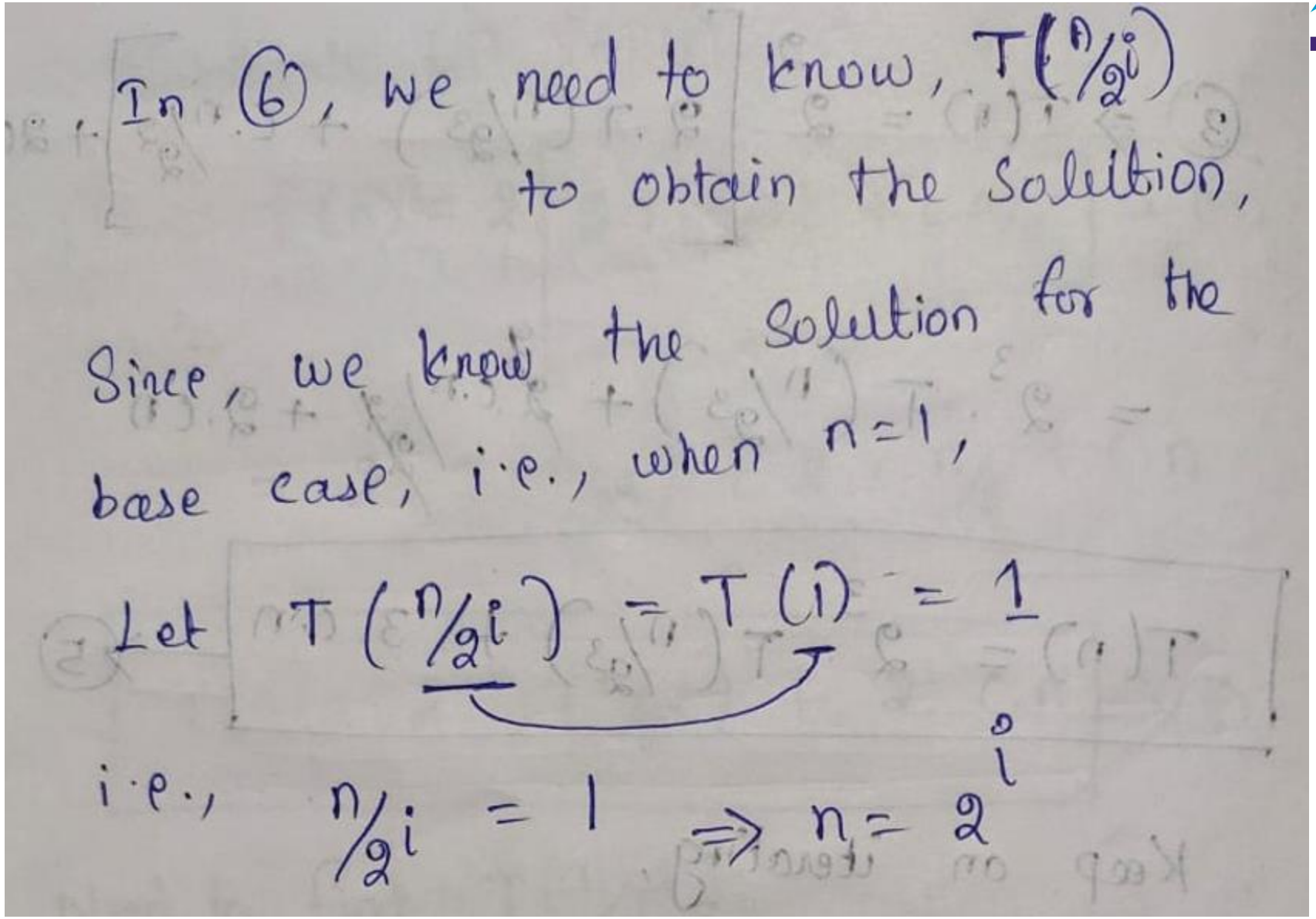
At the i^{th} Iteration,

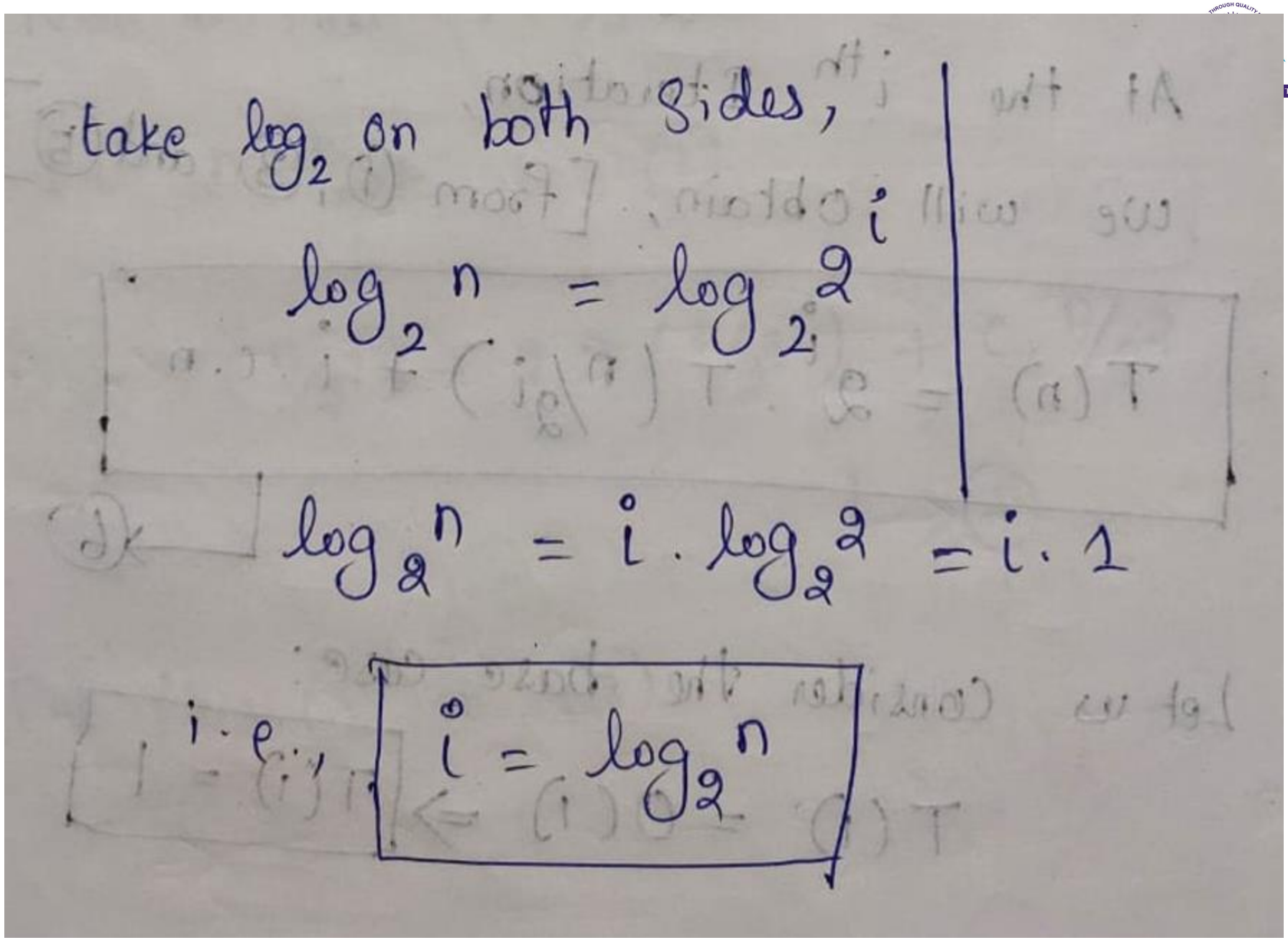
we will obtain, [from (1), (3) and (5)]

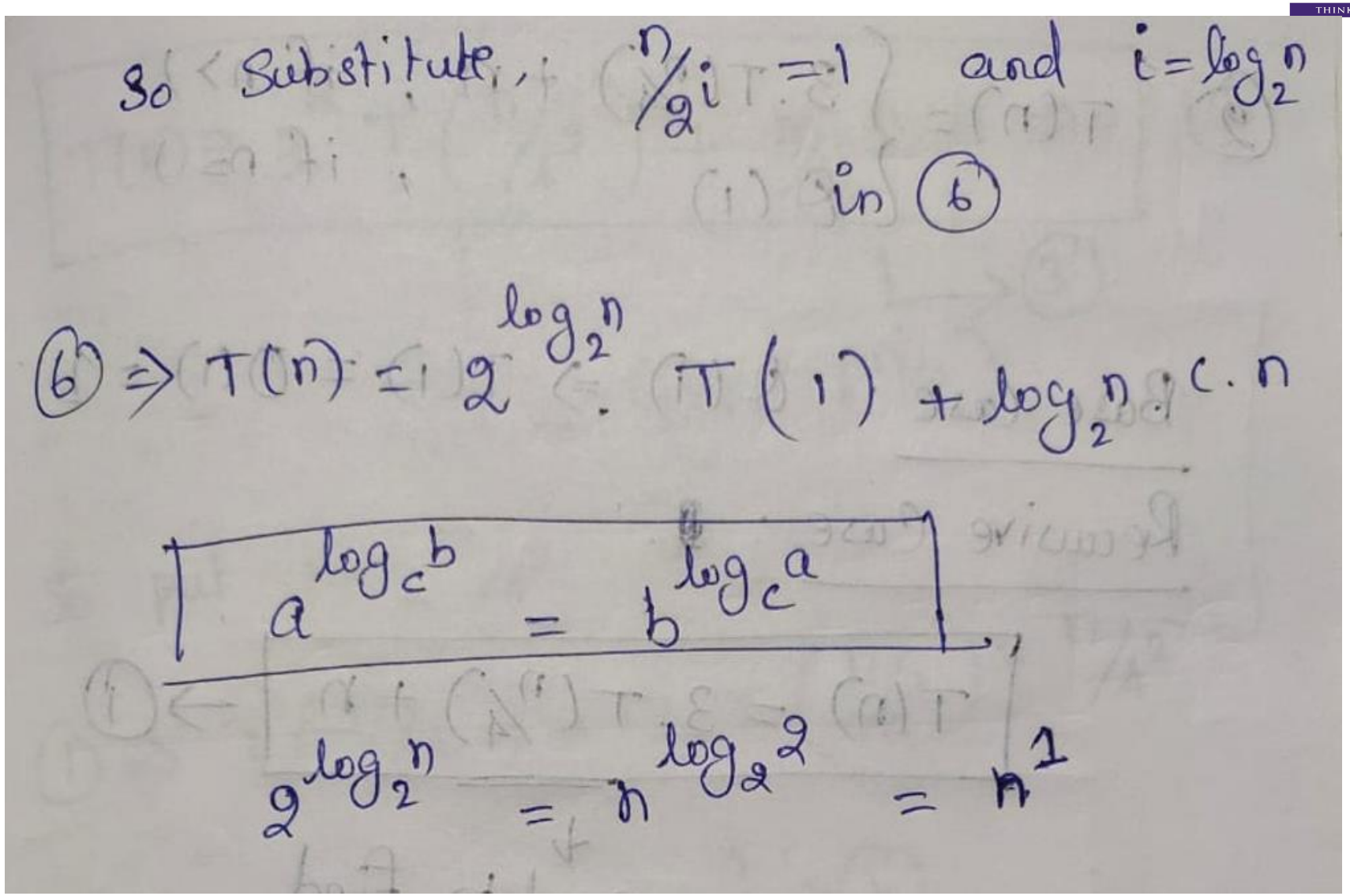
$$T(n) = 2^i \cdot T(n/2^i) + i \cdot c \cdot n \quad \rightarrow (6)$$

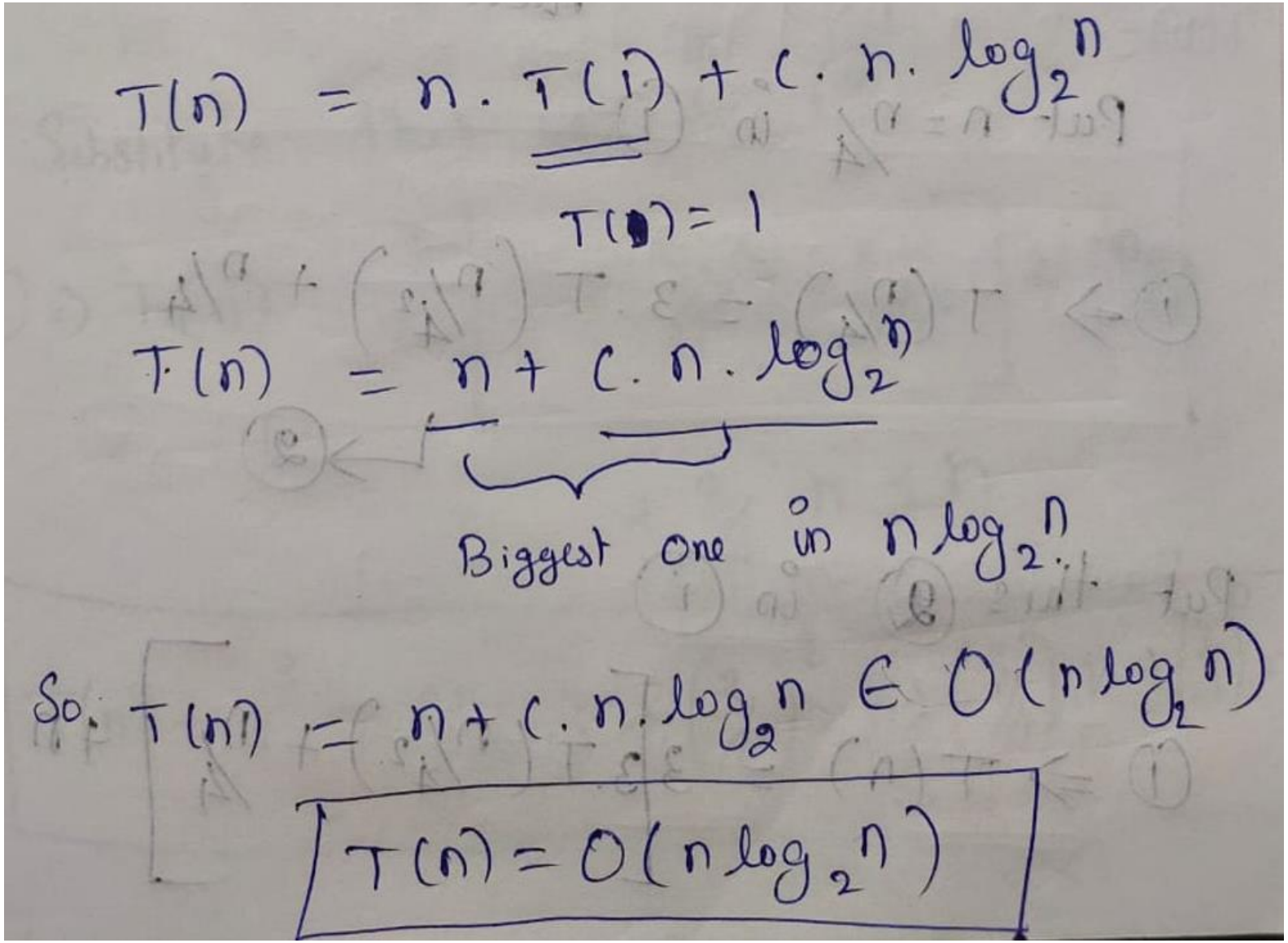
Let us consider the base case:

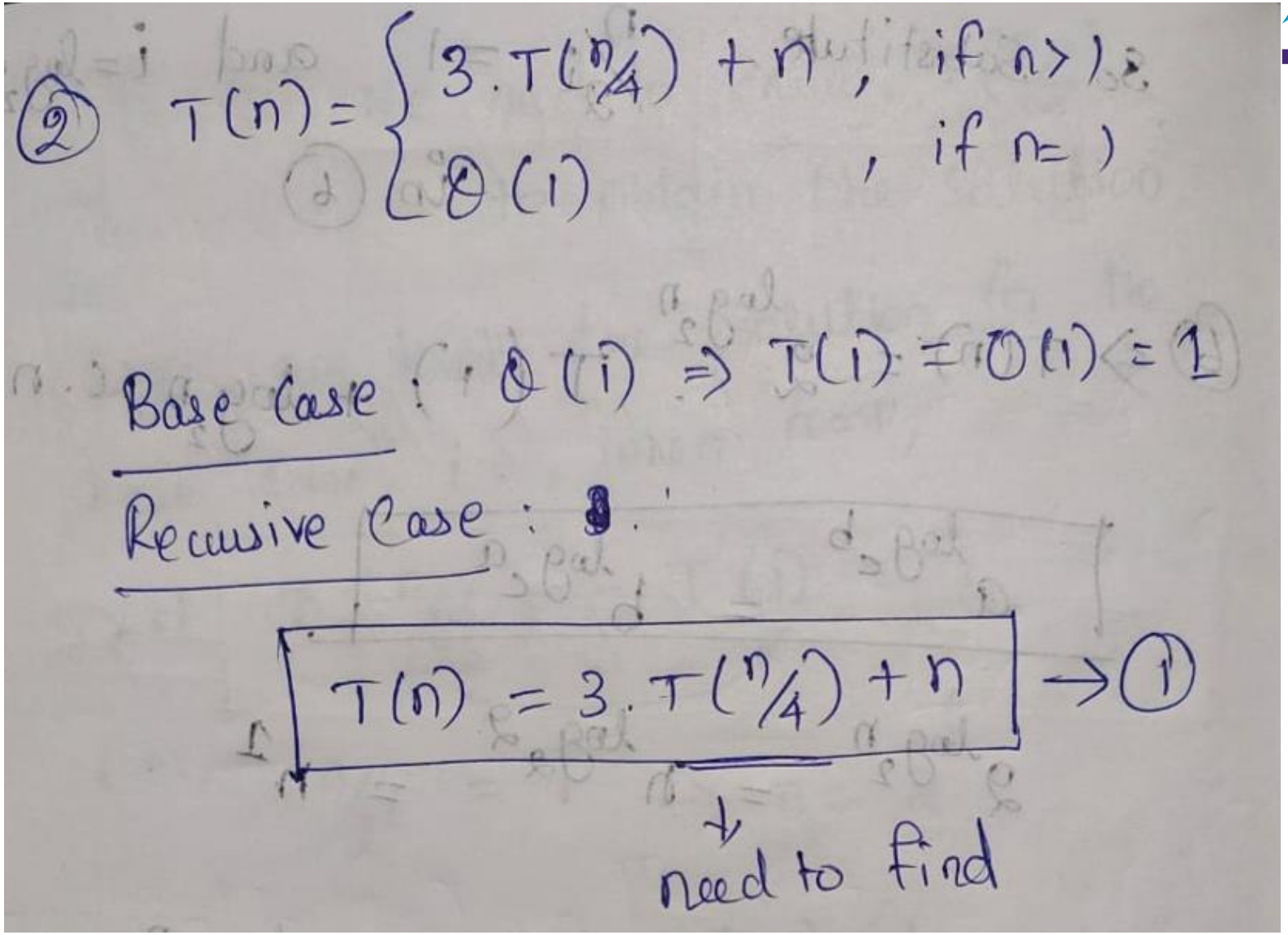
$$T(1) = O(1) \Rightarrow \boxed{T(1) = 1}$$


In (6), we need to know, T(n/2^i)
to obtain the Solution,
Since, we know the Solution for the
base case, i.e., when n=1,
Let T(n/2^i) = T(1) = 1
i.e., n/2^i = 1 ⇒ n = 2^i


take $\log_2$ on both sides;
$\log_2 n = \log_2 2^i$
$\log_2 n = i \cdot \log_2 2 = i \cdot 1$
$i = \log_2 n$


So Substitute, $\frac{n}{2^i} = 1$ and $i = \log_2 n$
in (b)
(b) $\Rightarrow T(n) = 2^{\log_2 n} \cdot T(1) + \log_2 n \cdot c \cdot n$
$a^{\log_c b} = b^{\log_c a}$
$2^{\log_2 n} = n^{\log_2 2} = n^1$


$T(n) = n \cdot T(1) + c \cdot n \cdot \log_2 n$
$T(1) = 1$
$T(n) = n + c \cdot n \cdot \log_2 n$
Biggest one in $n \log_2 n$
So, $T(n) = n + c \cdot n \cdot \log_2 n \in O(n \log_2 n)$
$T(n) = O(n \log_2 n)$

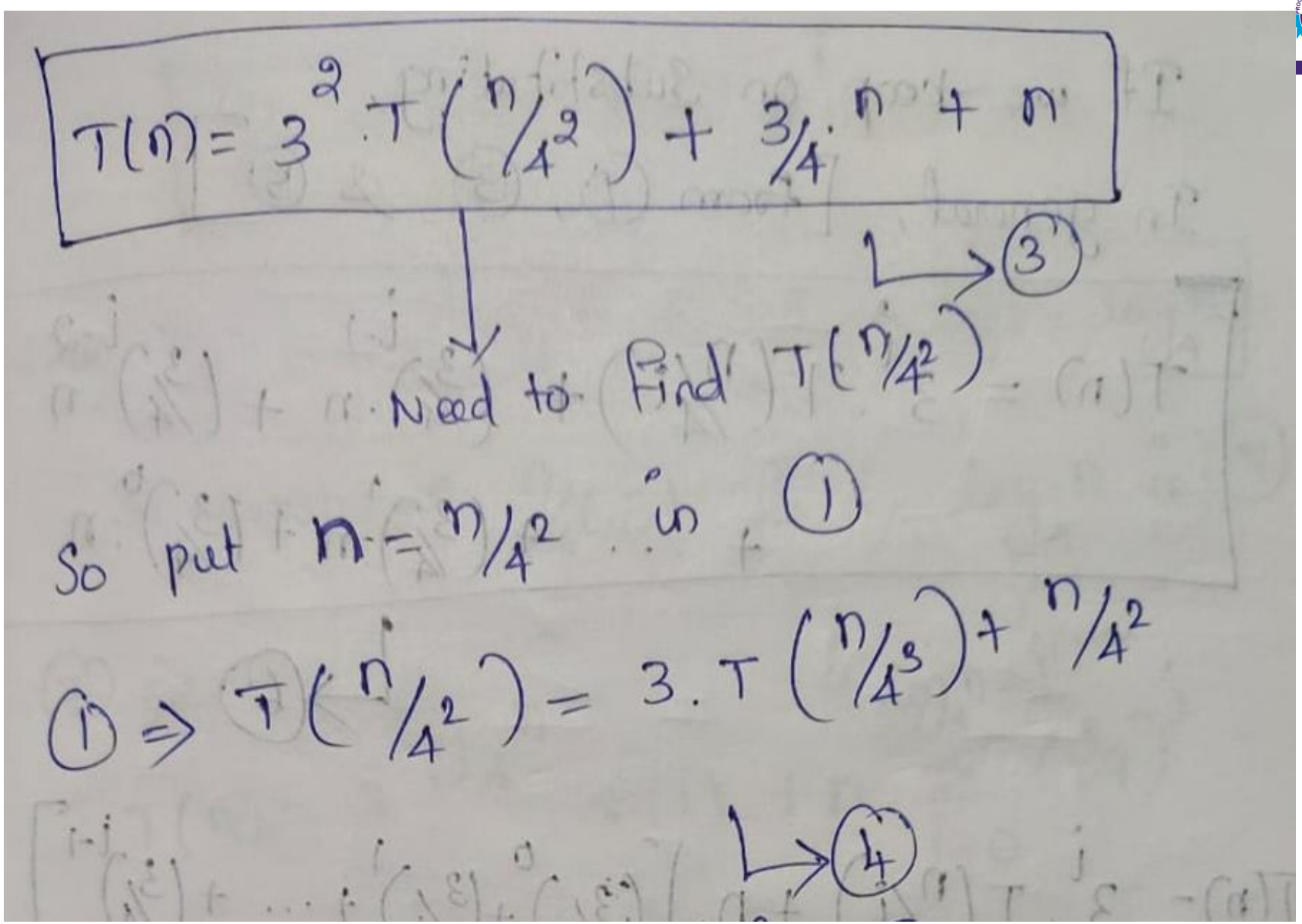

② $T(n) = \begin{cases} 3 \cdot T(n/4) + n, & \text{if } n > 1 \\ \Theta(1), & \text{if } n = 1 \end{cases}$
Base Case : $\Theta(1) \Rightarrow T(1) = \Theta(1) = 1$
Recursive Case :
$T(n) = 3 \cdot T(n/4) + n$ → ①
need to find

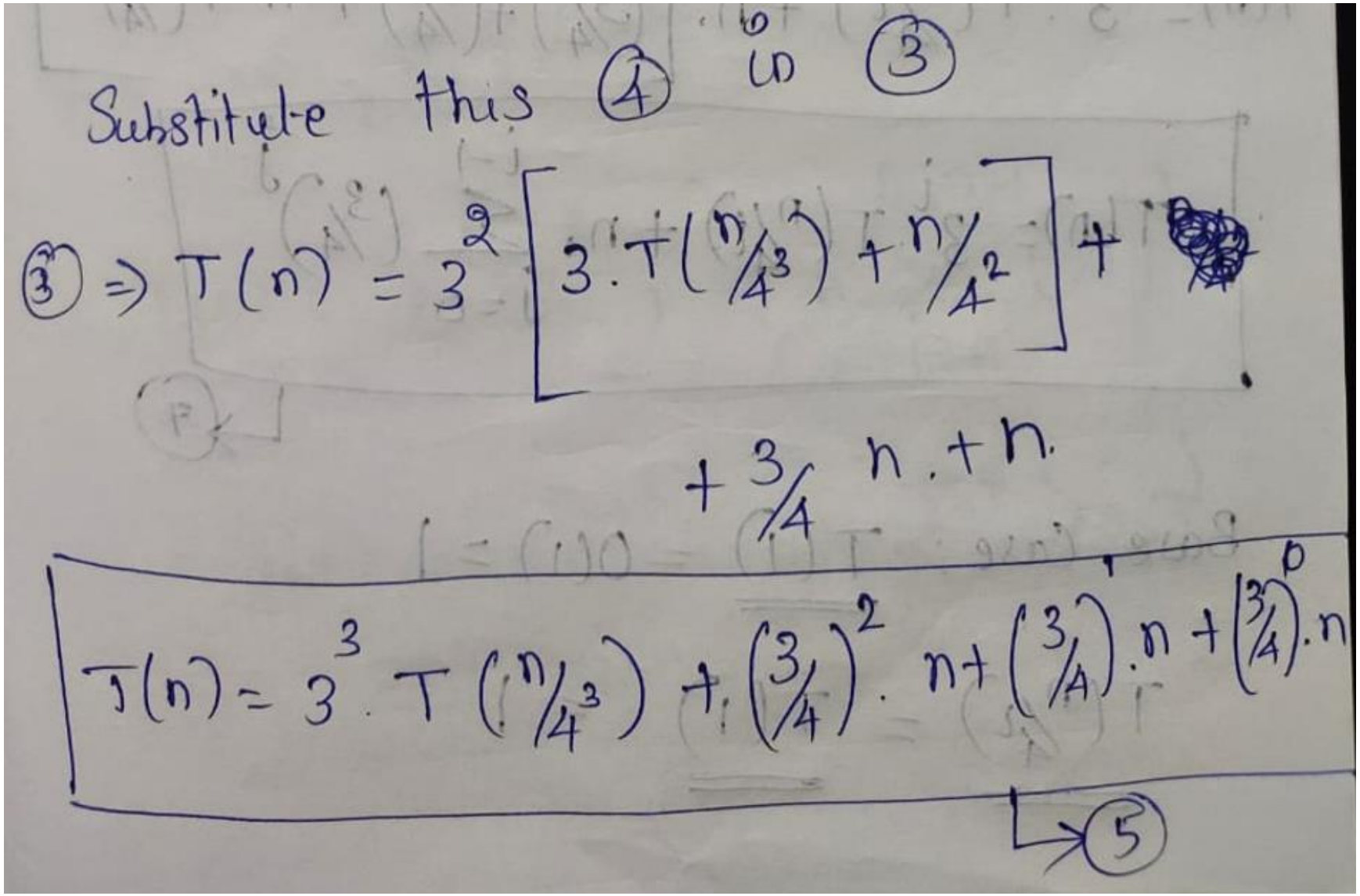
Put $n = n/4$ in (1)

(1) $\Rightarrow T(n/4) = 3 \cdot T(n/4^2) + n/4$ → (2)

Put this (2) in (1)

(1) $\Rightarrow T(n) = 3\left[3 \cdot T(n/4^2) + n/4\right] + n$

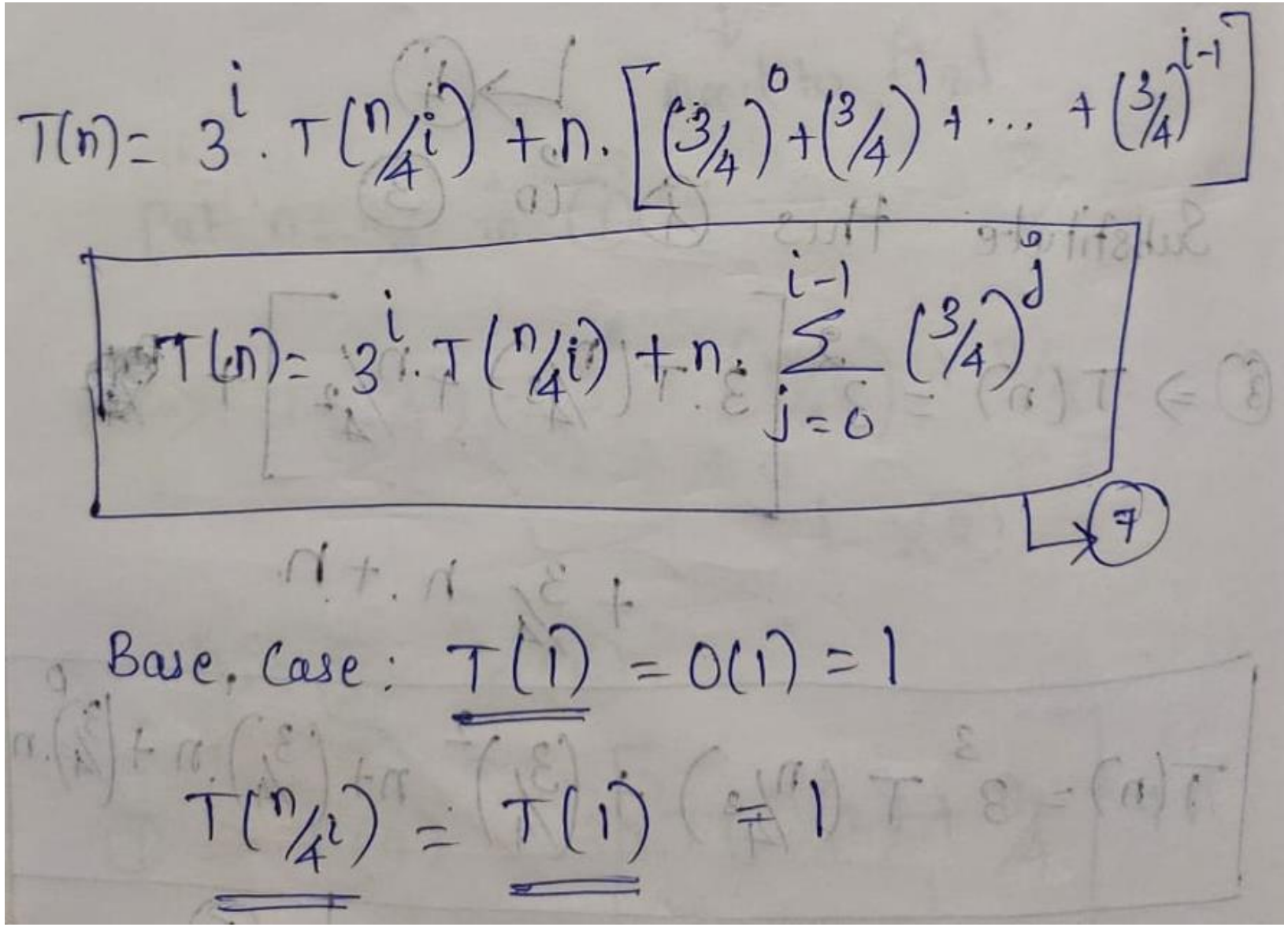

$T(n) = 3^2 \cdot T\left(n/4^2\right) + 3/4 \cdot n^2 + n$
Need to Find $T(n/4^2)$
So put $n = n/4^2$ in (1)
(1) $\Rightarrow T\left(n/4^2\right) = 3 \cdot T\left(n/4^3\right) + n/4^2$
(3)
(4)

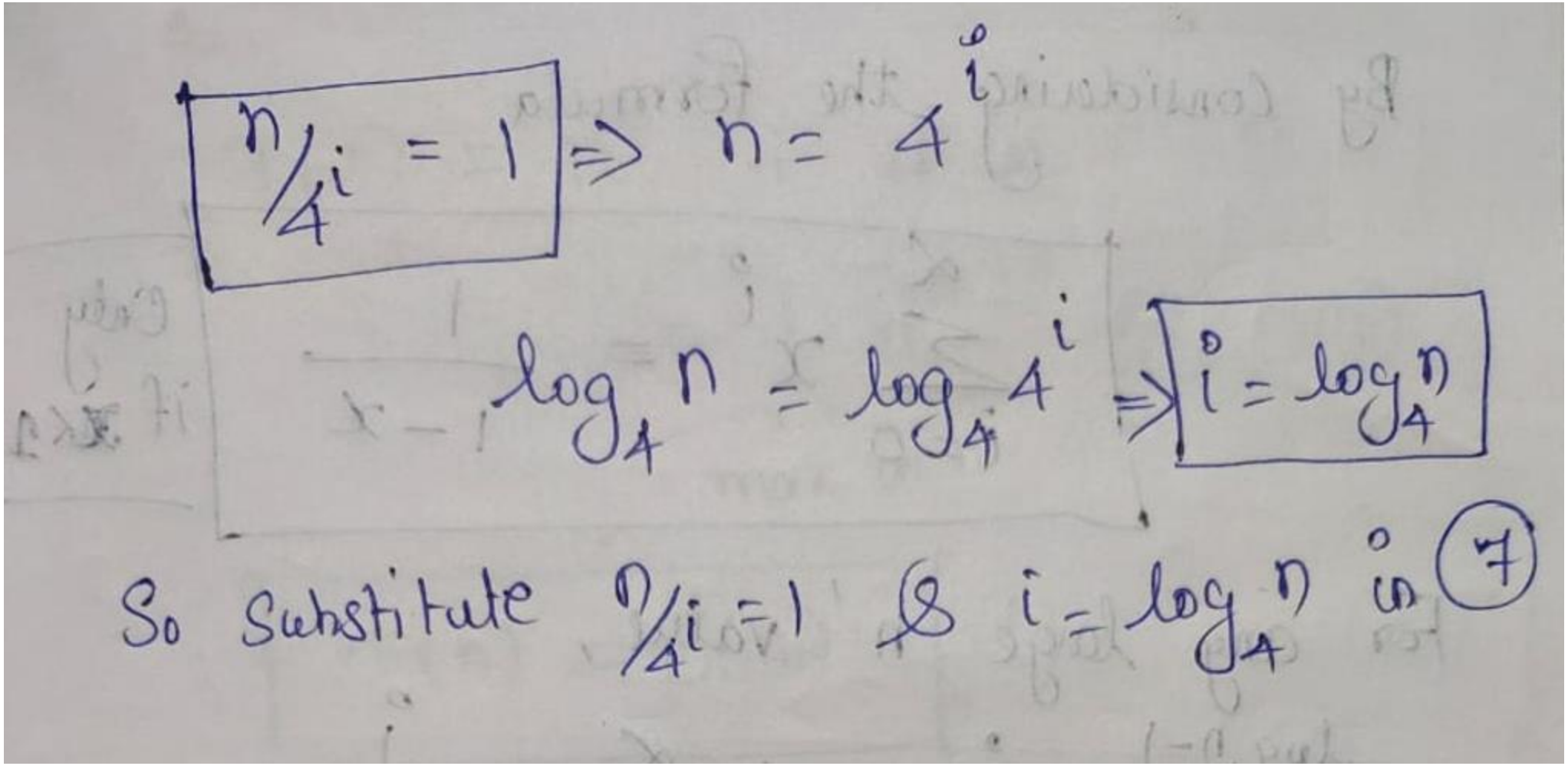

Substitute this (4) in (3)
(3) ⇒ T(n) = 3^2 [3.T(n/4^3) + n/4^2] +
+ 3/4 n . + n.
T(n) = 3^3 . T(n/4^3) + (3/4)^2 . n + (3/4)^1 . n + (3/4)^0 . n
5

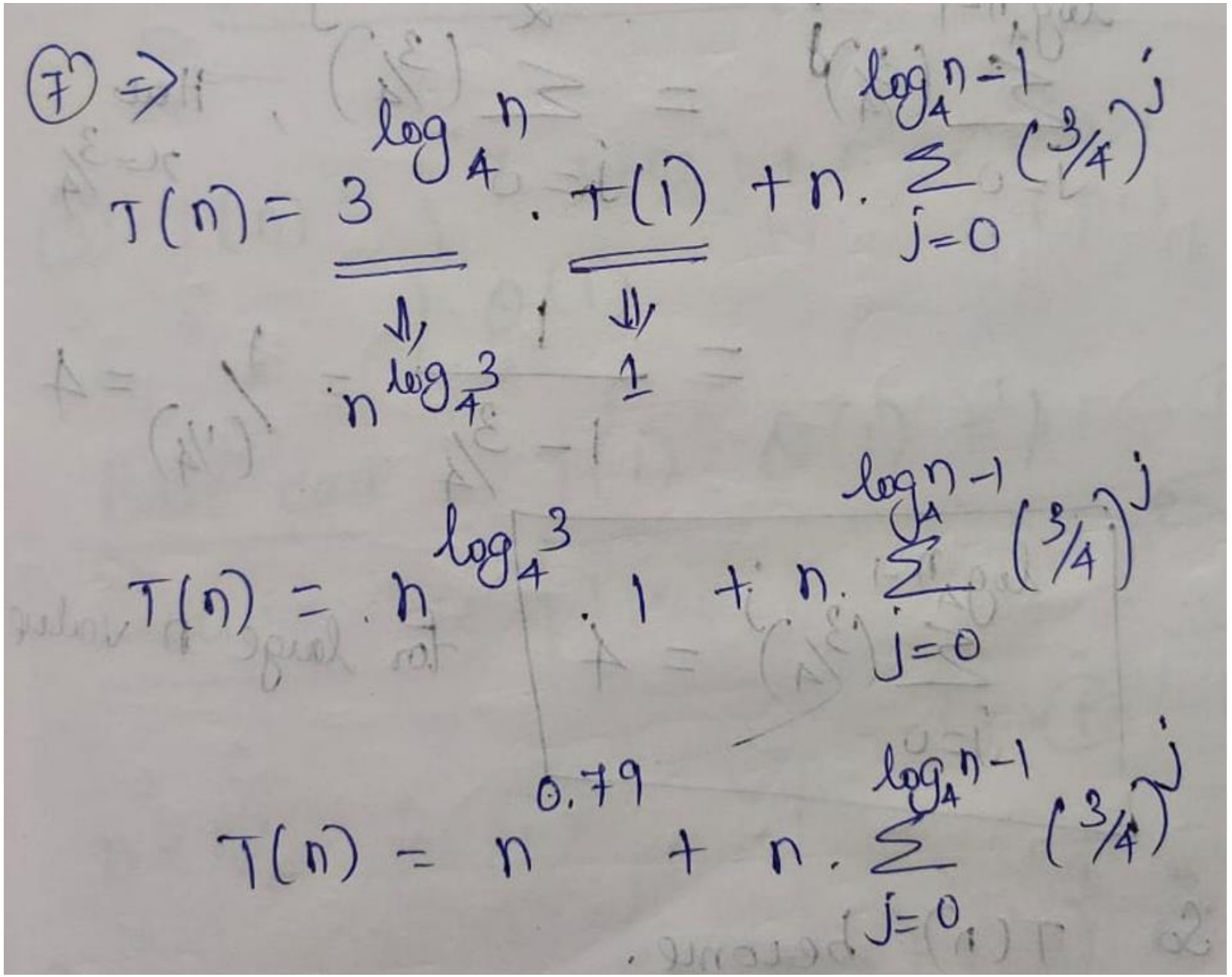
If we keep on substituting,

In general, [from ①, ③ & ⑤]

$$T(n) = 3^i \cdot T\left(\frac{n}{4^i}\right) + \left(\frac{3}{4}\right)^{i-1} \cdot n + \left(\frac{3}{4}\right)^{i-2} \cdot n + \ldots + \left(\frac{3}{4}\right)^{1} \cdot n + \left(\frac{3}{4}\right)^{0} \cdot n \quad \rightarrow ⑥$$


$T(n) = 3^i \cdot T(n/4^i) + n \cdot \left[(3/4)^0 + (3/4)^1 + \ldots + (3/4)^{i-1} \right]$
$T(n) = 3^i \cdot T(n/4^i) + n \cdot \sum_{j=0}^{i-1} (3/4)^j$
7
Base Case: $T(1) = O(1) = 1$
$T(n/4^i) = T(1) = 1$

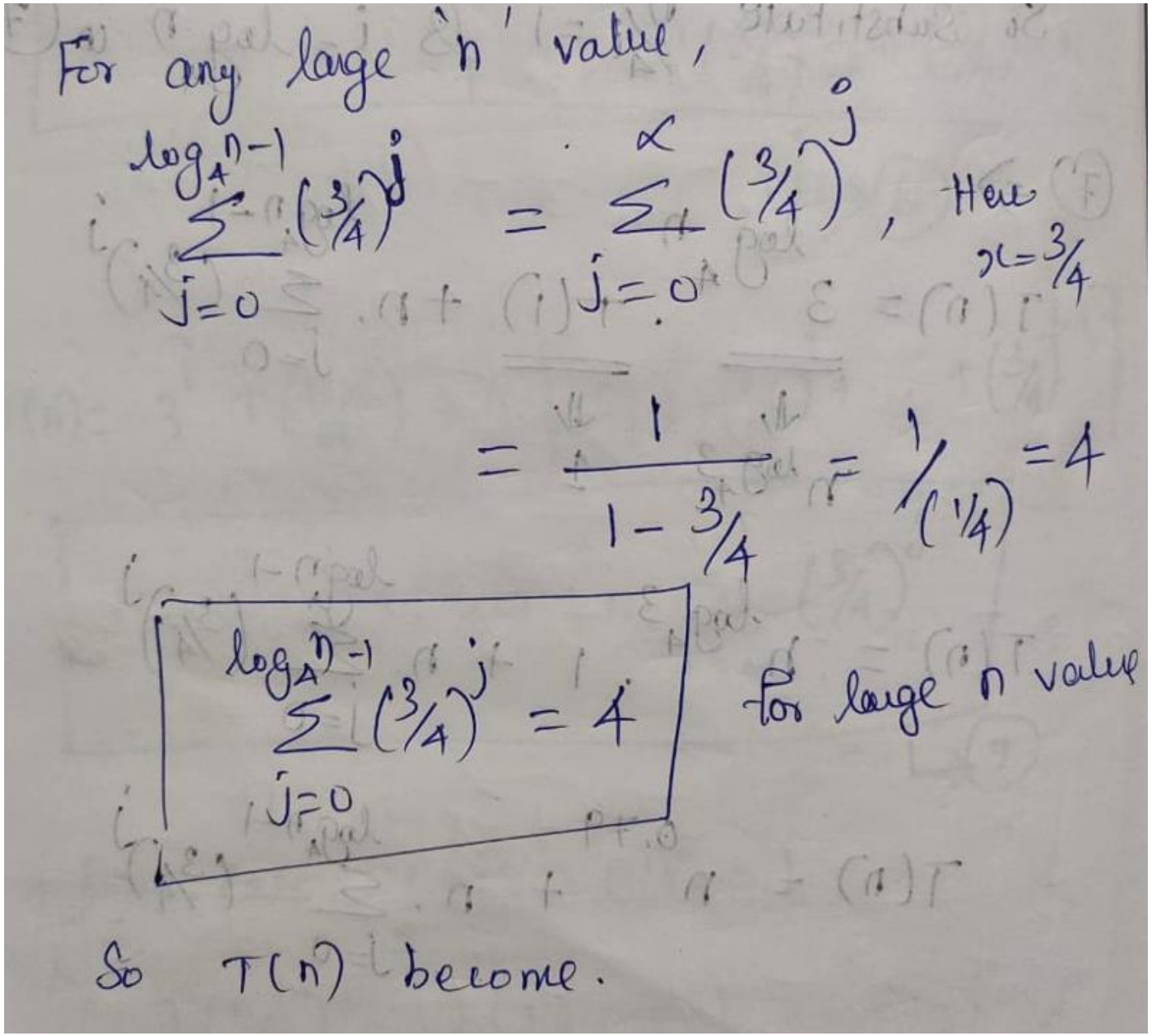

$n/4^i = 1 \Rightarrow n = 4^i$
$\log_4 n = \log_4 4^i \Rightarrow i = \log_4 n$
So Substitute $n/4^i = 1$ & $i = \log_4 n$ in (7)


(7) ⇒
$T(n) = 3^{\log_4 n} \cdot T(1) + n \cdot \sum_{j=0}^{\log_4 n - 1} (3/4)^j$
$n^{\log_4 3}$
1
$T(n) = n^{\log_4 3} \cdot 1 + n \cdot \sum_{j=0}^{\log_4 n - 1} (3/4)^j$
$T(n) = n^{0.79} + n \cdot \sum_{j=0}^{\log_4 n - 1} (3/4)^j$

By considering the formula

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

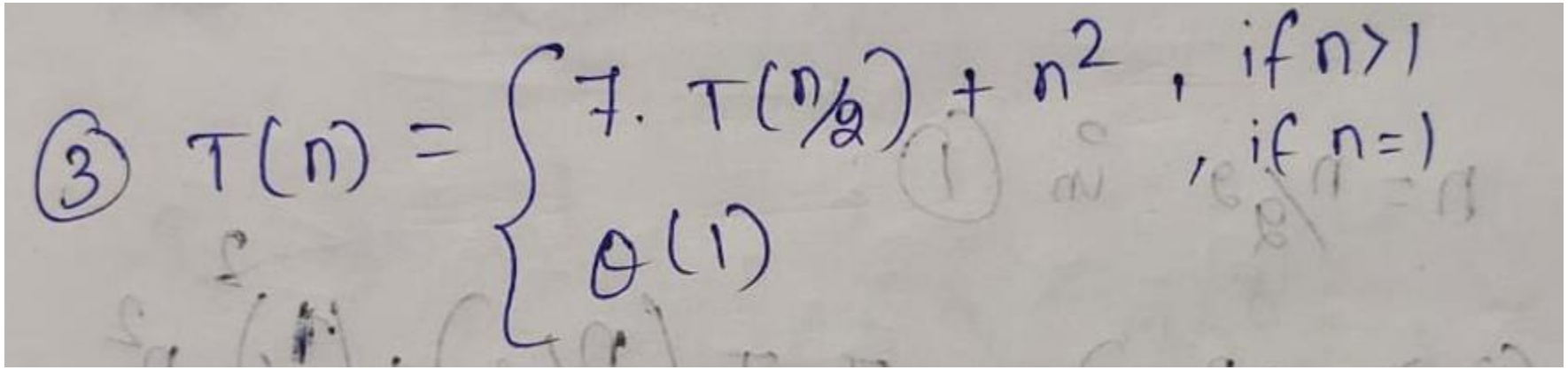
Only if $x < 1$

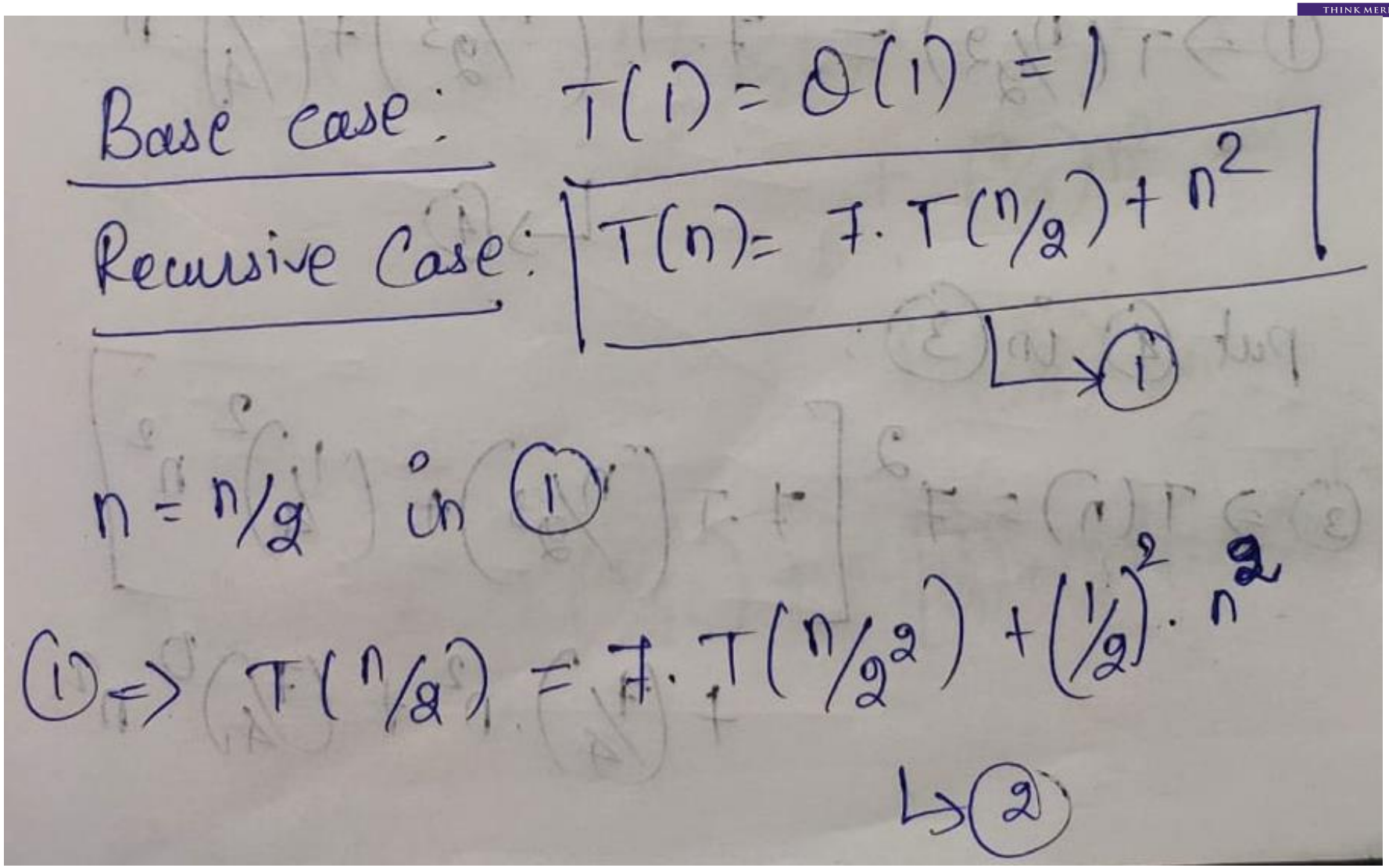

For any large 'n' value,
$\sum_{j=0}^{\log_4 n - 1} (3/4)^j = \sum_{j=0}^{\infty} (3/4)^j$, Here $x = 3/4$
$= \frac{1}{1 - 3/4} = 1/(1/4) = 4$
$\sum_{j=0}^{\log_4 n - 1} (3/4)^j = 4$
for large n value
So T(n) become.

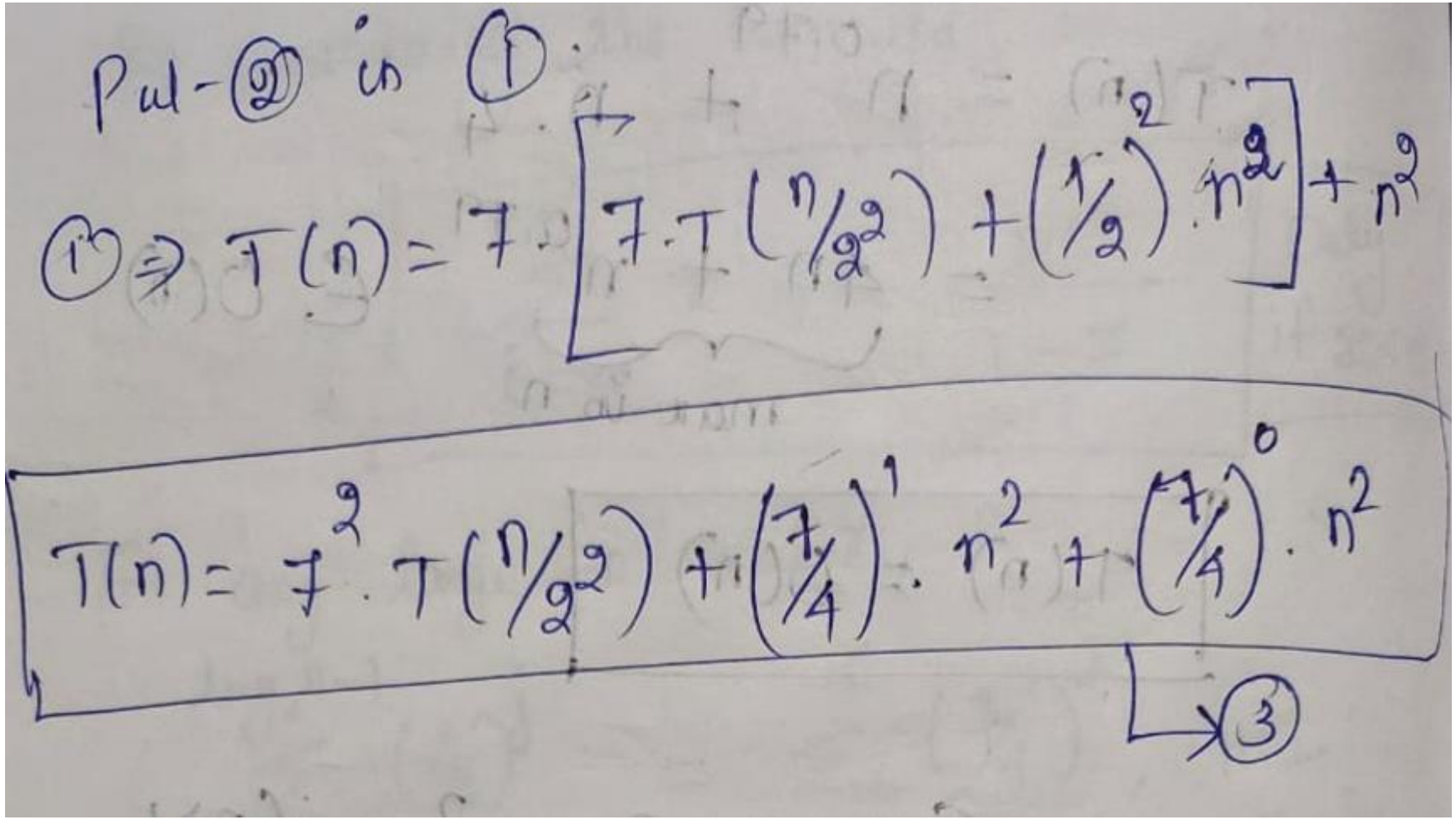
$$T(n) = n^{0.79} + n.4$$

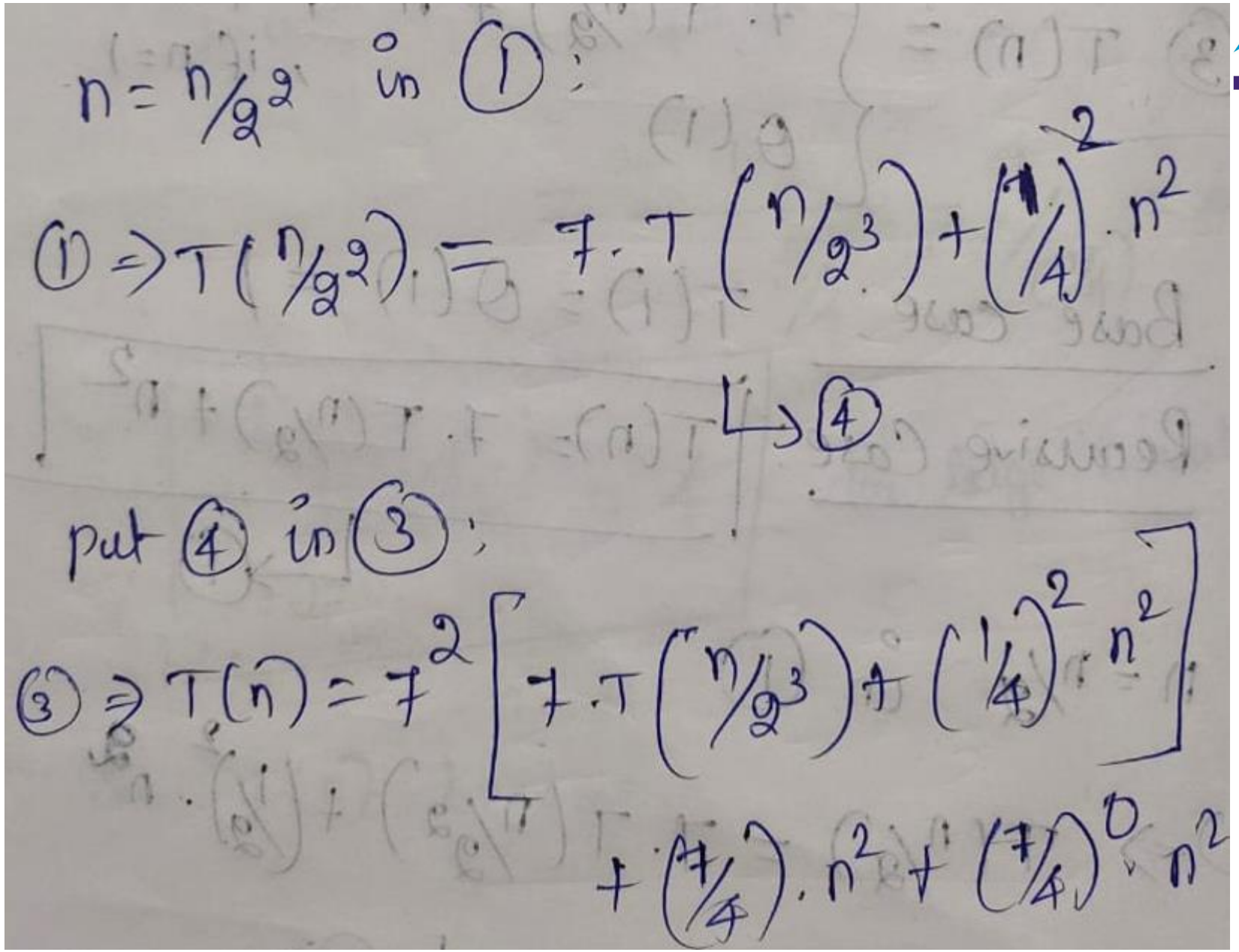
$$= \underbrace{4n + n^{0.79}}_{\max \text{ in } n} \in O(n)$$

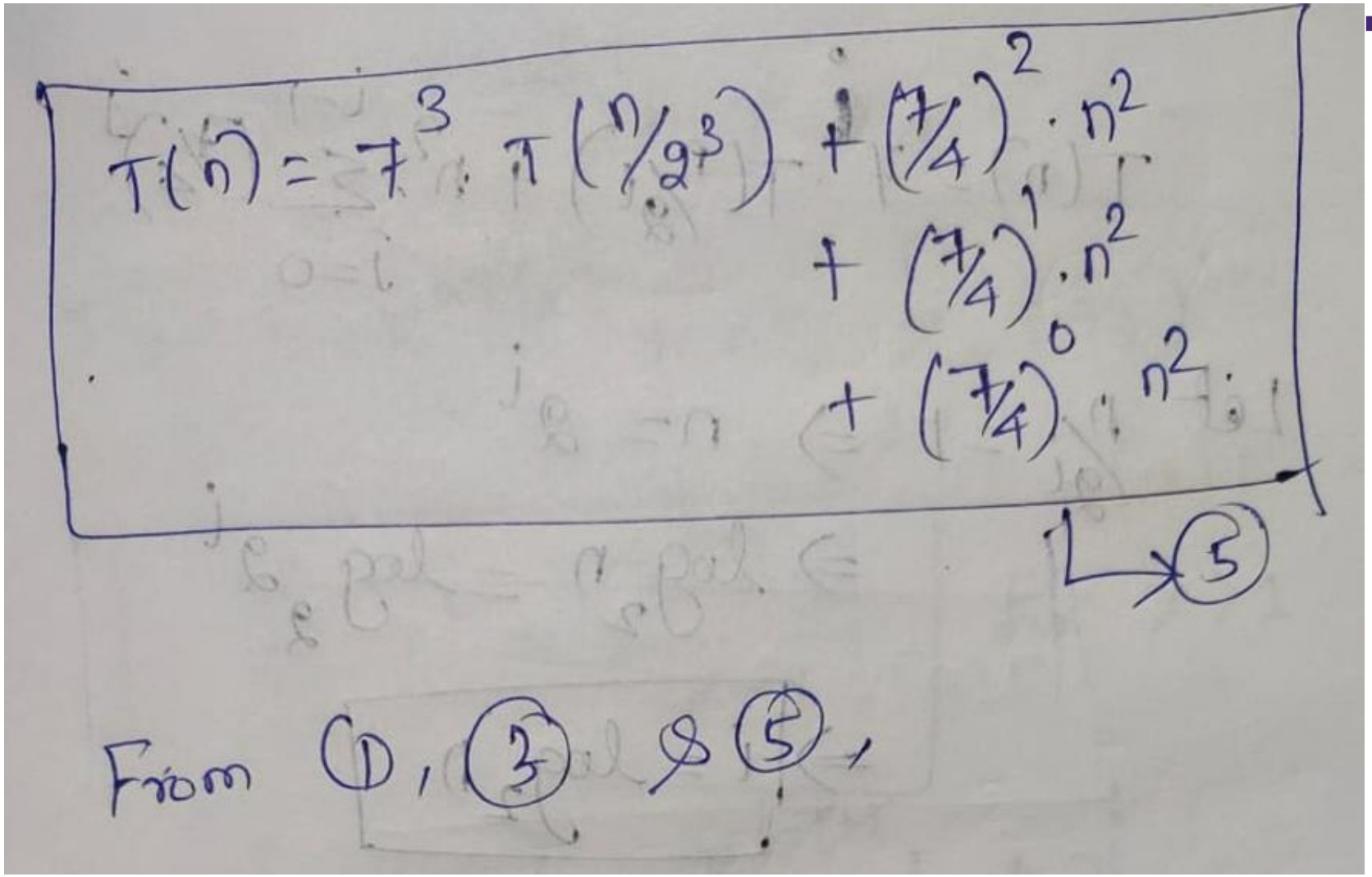
$$\boxed{T(n) = O(n)}$$


(3) T(n) = 7. T(n/2) + n^2 , if n>1
θ(1) , if n=1


Base case: T(1) = Θ(1) = 1
Recursive Case: T(n) = 7 · T(n/2) + n²
→ (1)
n = n/2 in (1)
(1) => T(n/2) = 7 · T(n/2²) + (1/2)² · n²
→ (2)


Put-② in ①:
① ⇒ T(n) = 7 · [7 · T(n/2²) + (1/2)² · n²] + n²
T(n) = 7² · T(n/2²) + (7/4)¹ · n² + (7/4)⁰ · n²
③

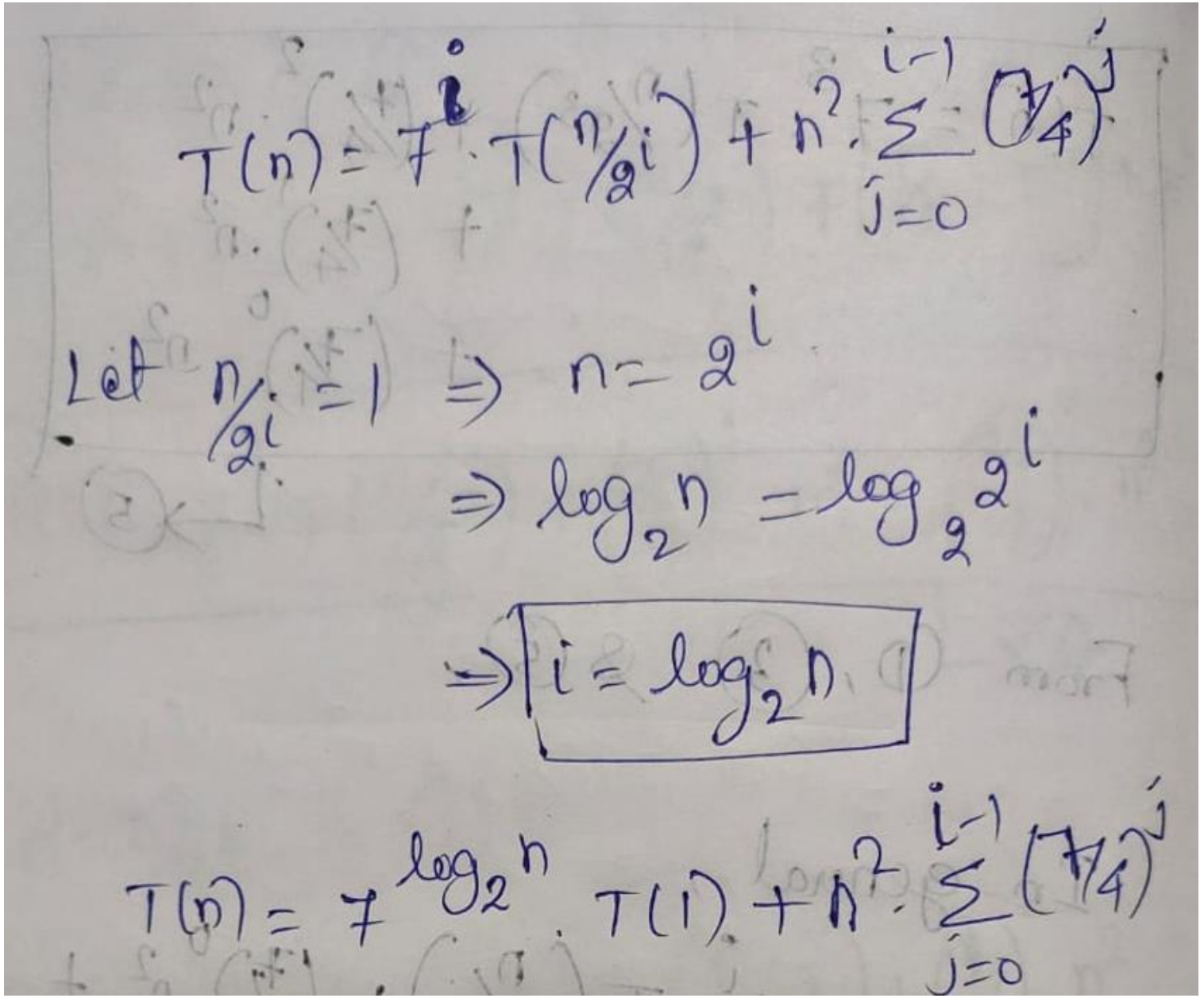

n = n/2^2 in (1):
(1) ⇒ T(n/2^2) = 7 . T(n/2^3) + (1/4)^2 . n^2
↳ (4)
put (4) in (3):
(3) ⇒ T(n) = 7^2 [7 . T(n/2^3) + (1/4)^2 . n^2]
+ (7/4) . n^2 + (7/4)^0 . n^2


$T(n) = 7^3 \cdot T(n/2^3) + (7/4)^2 \cdot n^2$
$+ (7/4)^1 \cdot n^2$
$+ (7/4)^0 \cdot n^2$
5
From (1), (3) & (5),

In general

$$T(n) = 7^i \cdot T\left(\frac{n}{2^i}\right) + \left(\frac{7}{4}\right)^0 \cdot n^2 + \left(\frac{7}{4}\right)^1 n^2 + \cdots + \left(\frac{7}{4}\right)^{i-1} \cdot n^2$$

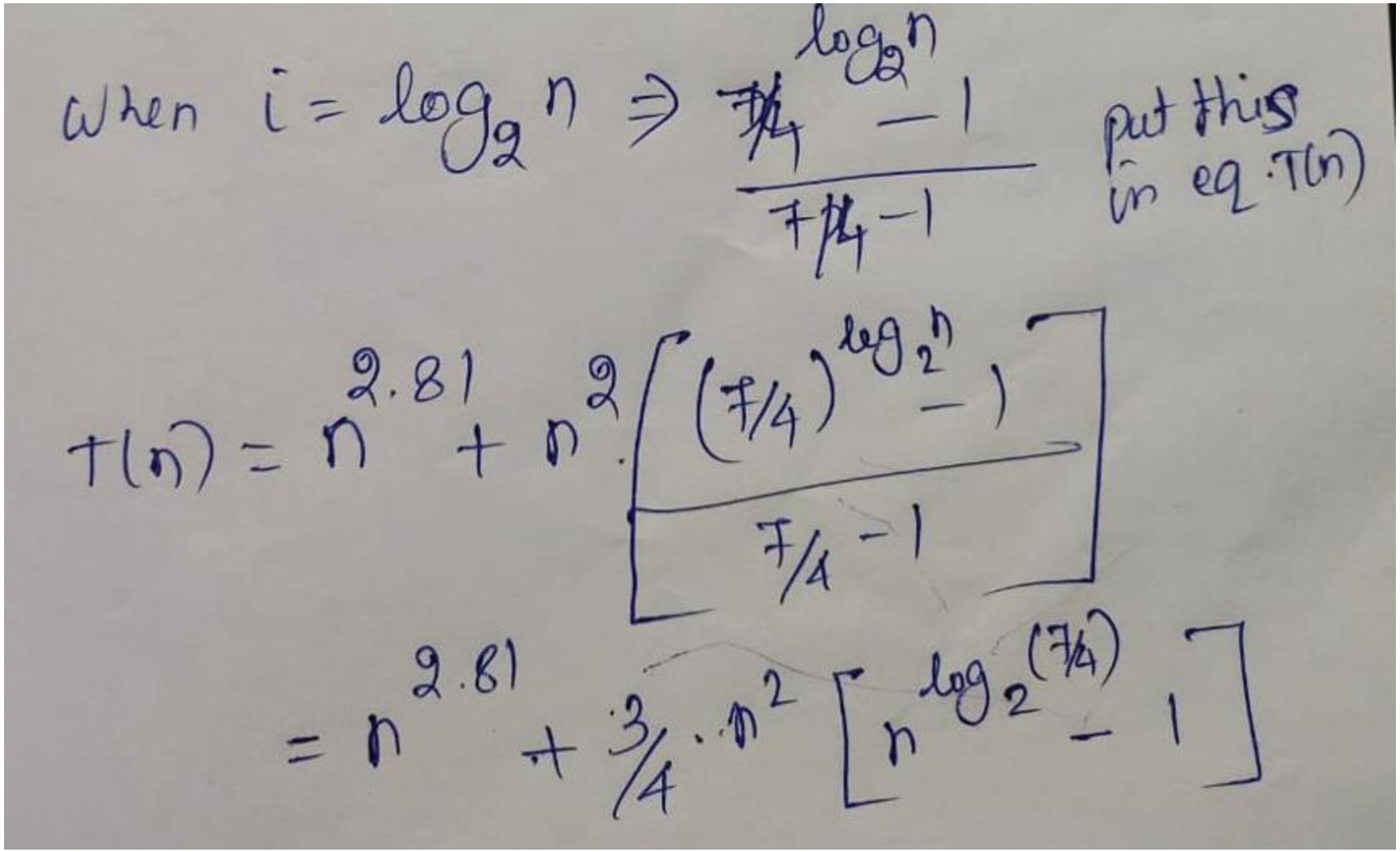
$$T(n) = 7^i \cdot T\left(\frac{n}{2^i}\right) + n^2 \left[\left(\frac{7}{4}\right)^0 + \left(\frac{7}{4}\right)^1 + \cdots + \left(\frac{7}{4}\right)^{(i-1)}\right]$$

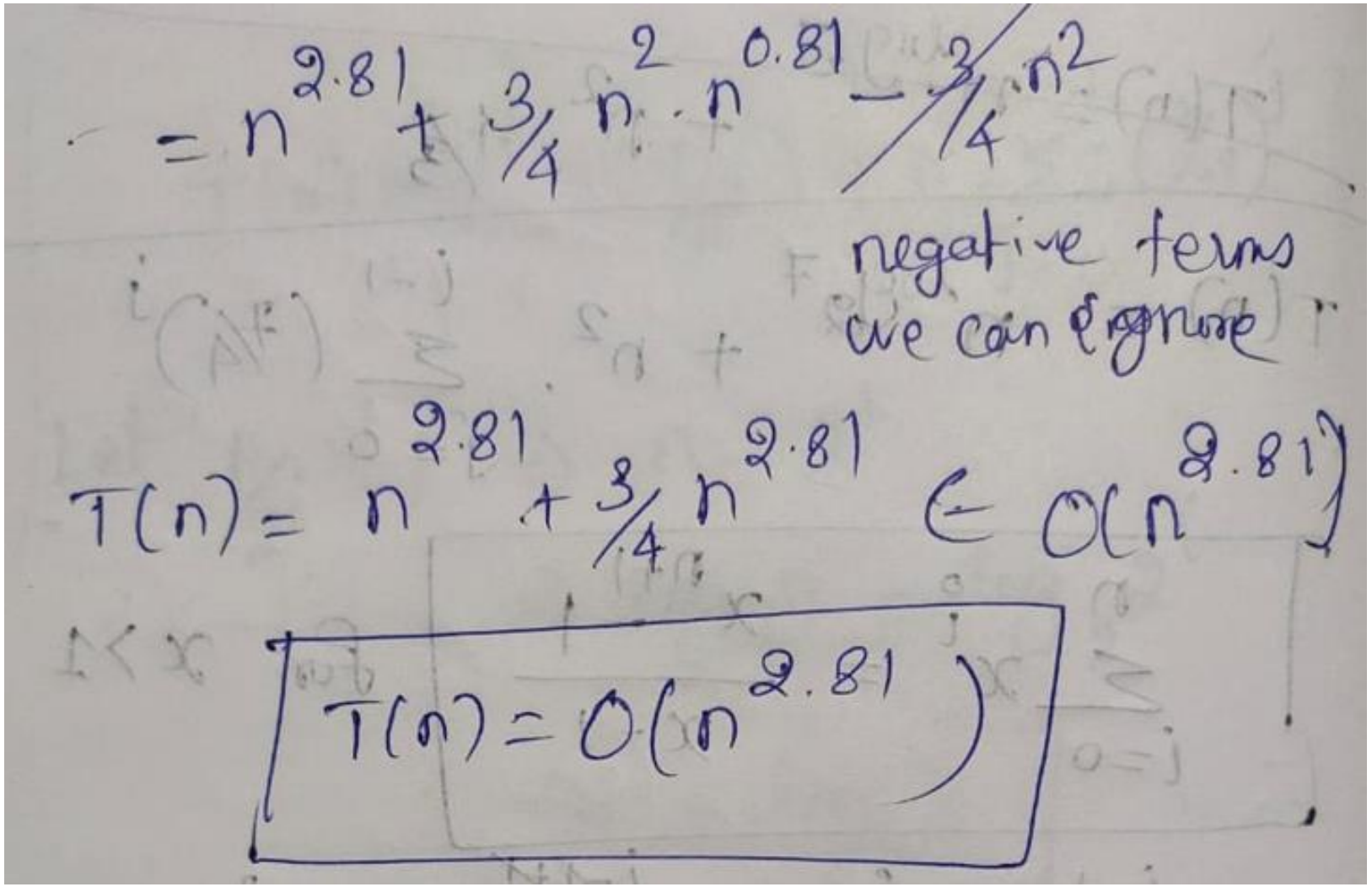

$T(n) = 7^i \cdot T(n/2^i) + n^2 \cdot \sum_{j=0}^{i-1} (7/4)^j$
Let $n/2^i = 1 \Rightarrow n = 2^i$
$\Rightarrow \log_2 n = \log_2 2^i$
$\Rightarrow i = \log_2 n$
$T(n) = 7^{\log_2 n} \cdot T(1) + n^2 \cdot \sum_{j=0}^{i-1} (7/4)^j$

$$T(n) = n^{\log_2 7} + n^2 \cdot \sum_{j=0}^{i-1} (7/4)^j$$

$$\boxed{\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}} \quad \text{for } x > 1$$

$$\sum_{j=0}^{i-1} (7/4)^j = \frac{(7/4)^{i-1+1} - 1}{7/4 - 1} = \frac{(7/4)^i - 1}{(7/4) - 1}$$


When $i = \log_2 n \Rightarrow \frac{7^{\log_2 n} - 1}{7/4 - 1}$ put this in eq. T(n)
$T(n) = n^{2.81} + n^2 \left[\frac{(7/4)^{\log_2 n} - 1}{7/4 - 1} \right]$
$= n^{2.81} + \frac{3}{4} \cdot n^2 \left[n^{\log_2 (7/4)} - 1 \right]$


$= n^{2.81} + \frac{3}{4} n^2 \cdot n^{0.81} - \frac{3}{4} n^2$
negative terms we can ignore
$T(n) = n^{2.81} + \frac{3}{4} n^{2.81} \in O(n^{2.81})$
$T(n) = O(n^{2.81})$

Solving Recurrences - Recursion Tree Method

Recursion Tree Method:

(For Solving Recurrences)

Examples:

(1) $$T(n) = \begin{cases} 2 \cdot T(n/2) + c \cdot n, & \text{if } n > 1 \\ \theta(1), & \text{if } n = 1 \end{cases}$$

No. of Sub problems $= 2 \Rightarrow a = 2$

Sub problem Size $= n/2 \Rightarrow b = 2$

Recursive Case:

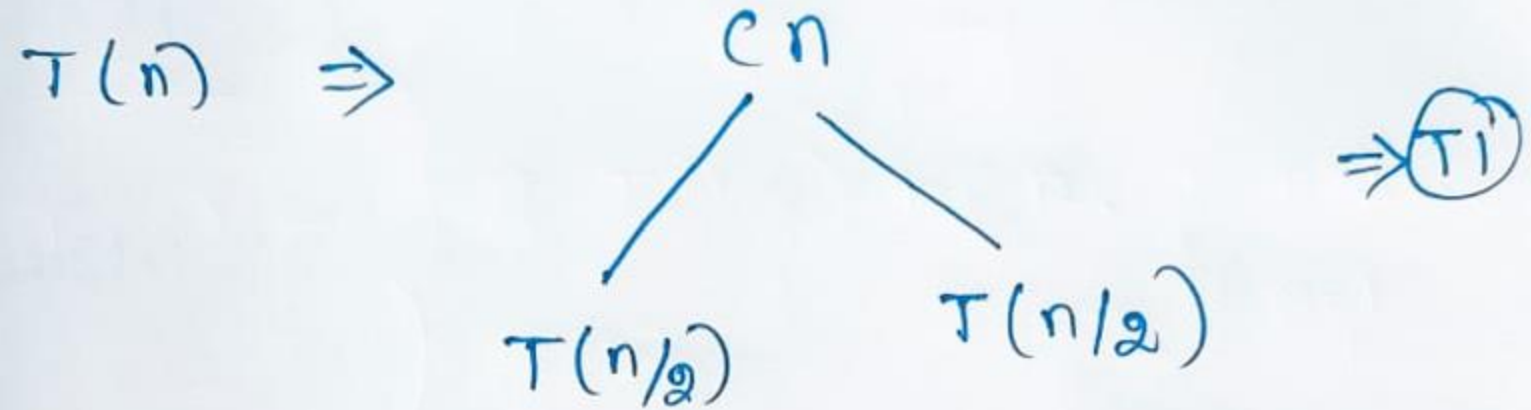
$$T(n) = 2 \cdot T(n/2) + c \cdot n \quad \rightarrow (1)$$

↓

→ Complexity for divide & Conquer.

→ Represent $T(n)$ as ~~binary~~ Tree with Root as $c \cdot n$ & the 2 children are $T(n/2)$

→ The no. of children is the no. of sub problems.



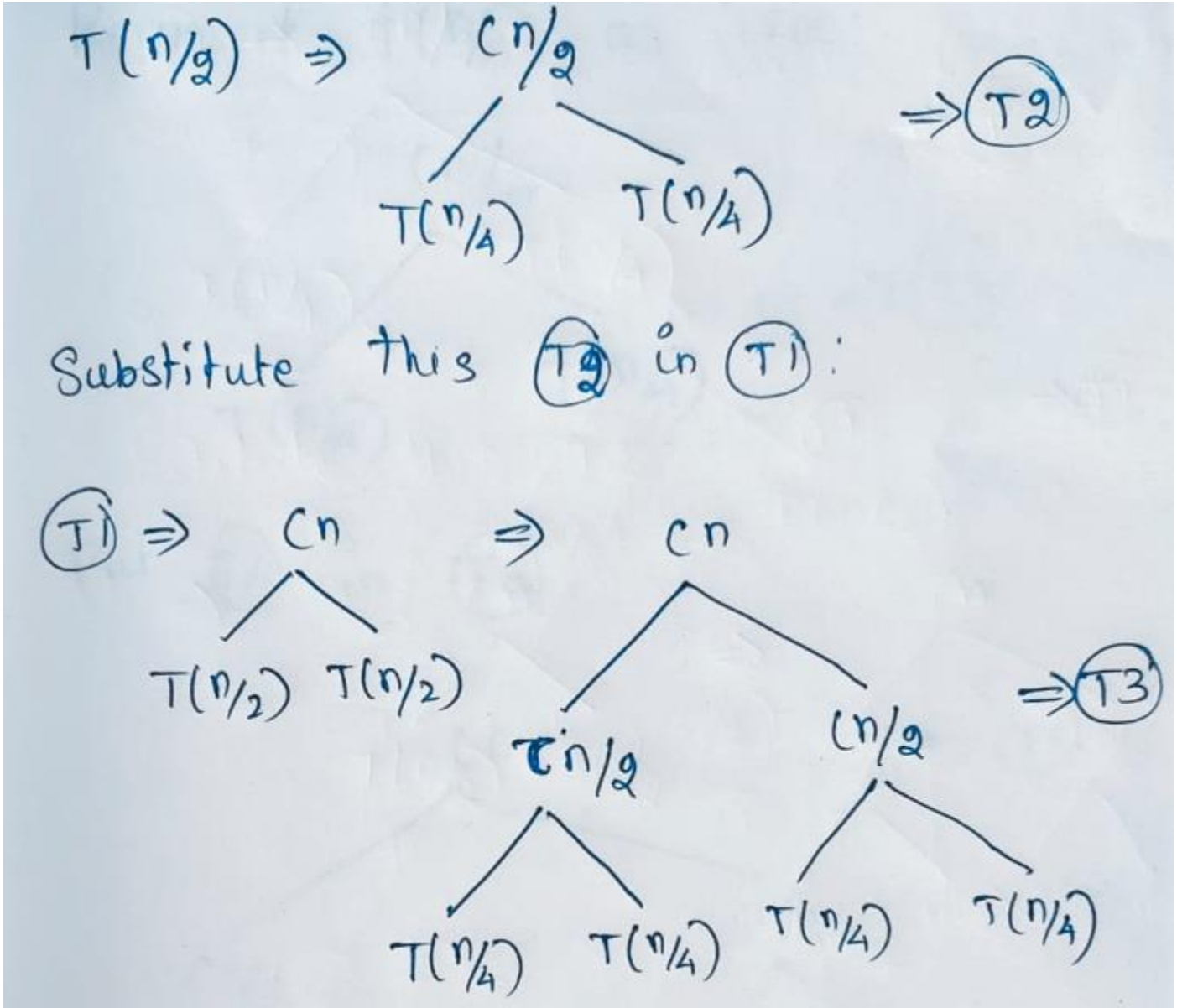
Now, Need to find $T(n/2)$:

Substitute $n = n/2$ in (1);

(1) $\Rightarrow T(n/2) = 2 \cdot T\left(\frac{n/2}{2}\right) + c \cdot n/2$

$$T(n/2) = 2 \cdot T(n/4) + c \cdot n/2 \quad \rightarrow (2)$$

Now Represent this (2) as Tree:


T(n/2) ⇒ cn/2
T(n/4) T(n/4)
⇒ T2
Substitute this T2 in T1 :
T1 ⇒ cn ⇒ cn
T(n/2) T(n/2)
⇒ T3
cn/2 cn/2
T(n/4) T(n/4) T(n/4) T(n/4)

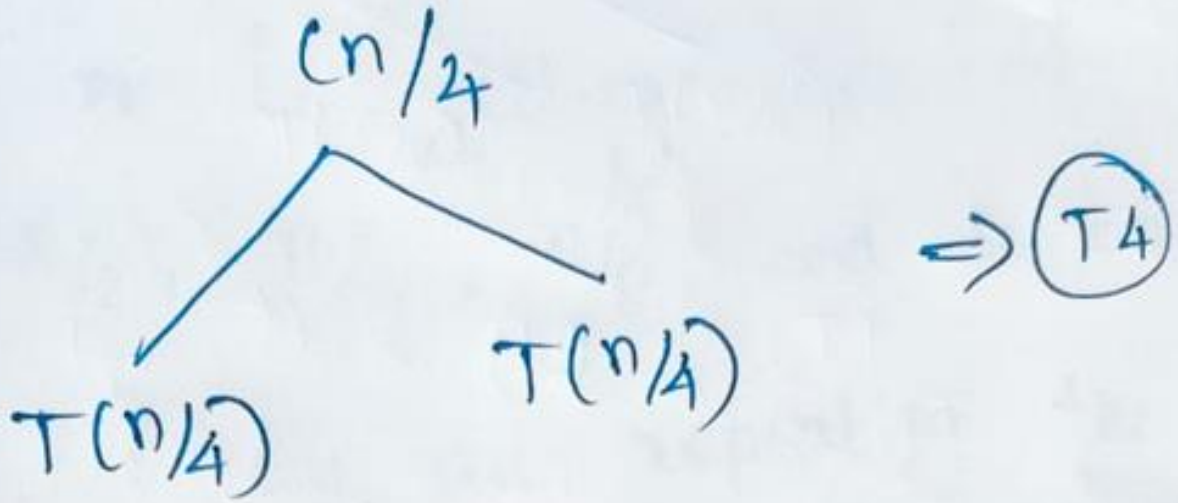
Now Need to find $T(n/4)$:

$n = n/4$ in (1)

(1) $\Rightarrow T(n/4) = 2 \cdot T\left(\frac{n/4}{2}\right) + c \cdot n/4$

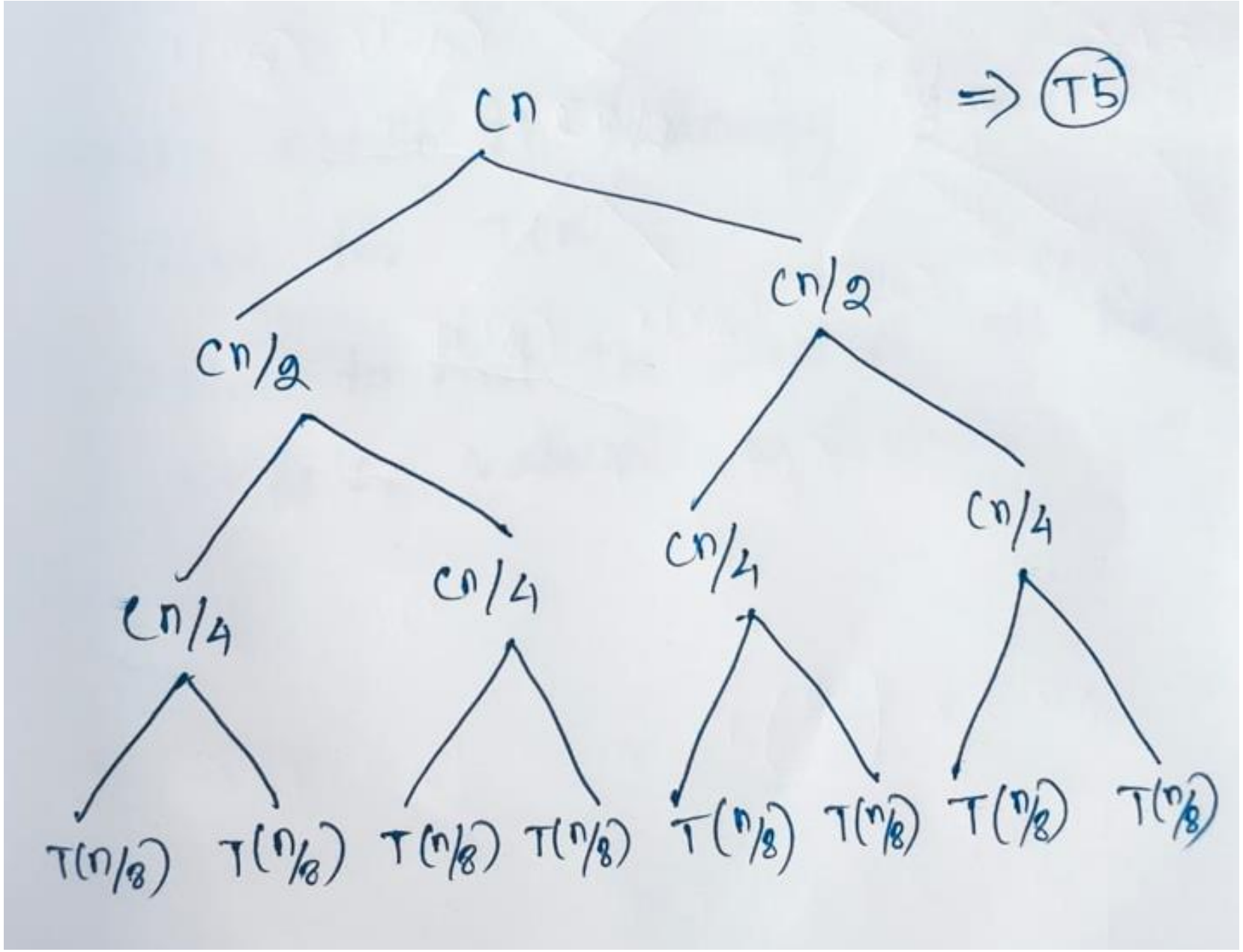
$$T(n/4) = 2 \cdot T(n/8) + cn/4 \quad \rightarrow (3)$$

Represent $T(n/4)$ as Tree:



$\Rightarrow$ (T4)

Put (T4) in (T3):

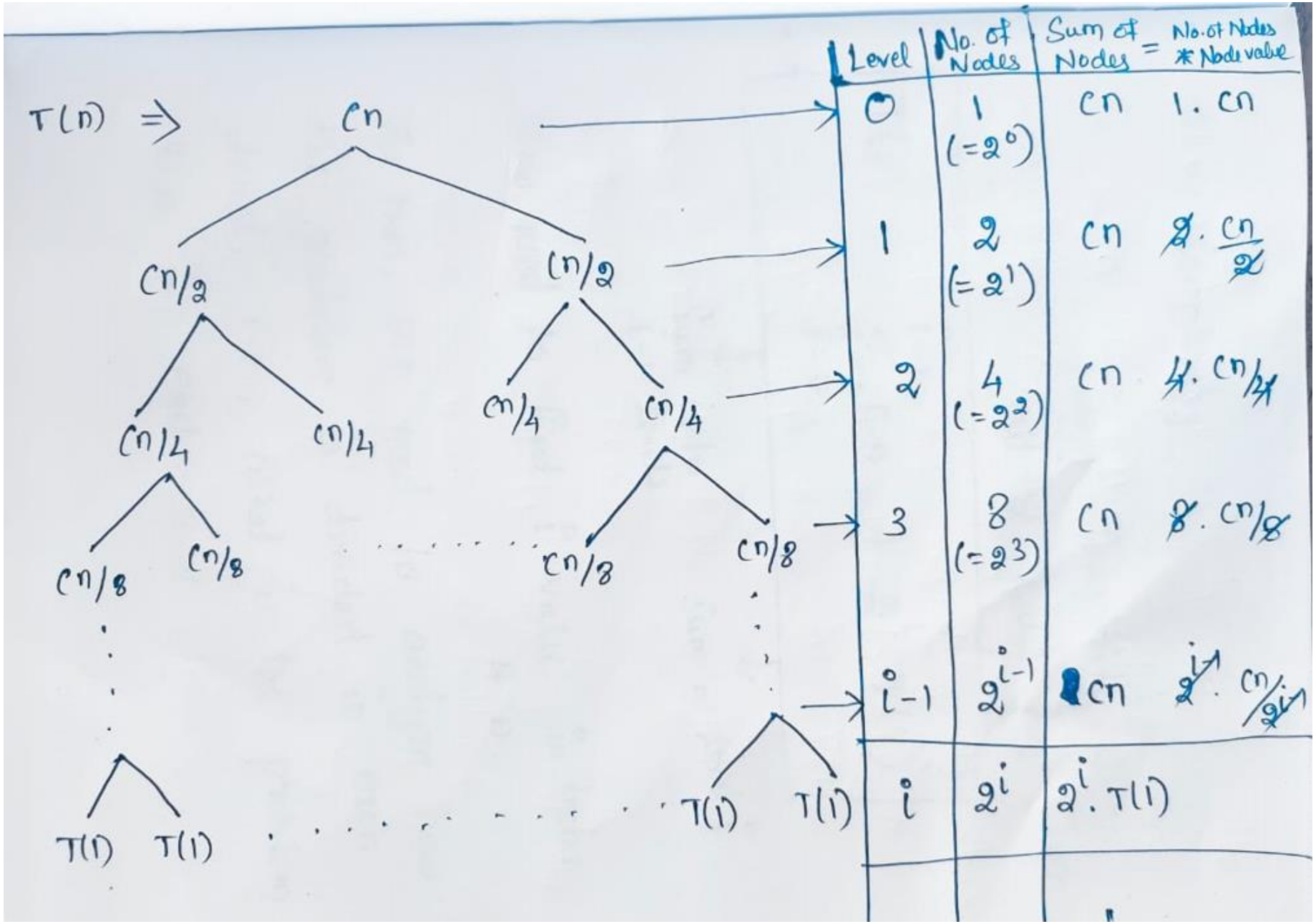

cn
⇒ T5
cn/2
cn/2
cn/4
cn/4
cn/4
cn/4
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)

→ In each level of the tree, the size n is getting reduced. as n, n/2, n/4, n/8 and so on.

→ So, if we are repeating the steps, once we will get a tree with all the leaf nodes are T(1).

→ We obtain the following generic Tree for T(n).

→ Need to find the sum of all the node's values. as follows.


T(n) ⇒
cn
cn/2
cn/2
cn/4
cn/4
cn/4
cn/4
cn/8
cn/8
cn/8
cn/8
T(1)
T(1)
T(1)
T(1)
Level
No. of Nodes
Sum of Nodes = No. of Nodes * Node value
0
1 (=2^0)
cn
1. cn
1
2 (=2^1)
cn
2. cn/2
2
4 (=2^2)
cn
4. cn/4
3
8 (=2^3)
cn
8. cn/8
i-1
2^(i-1)
cn
2^i . cn/2^i
i
2^i
2^i . T(1)

Time Complexity

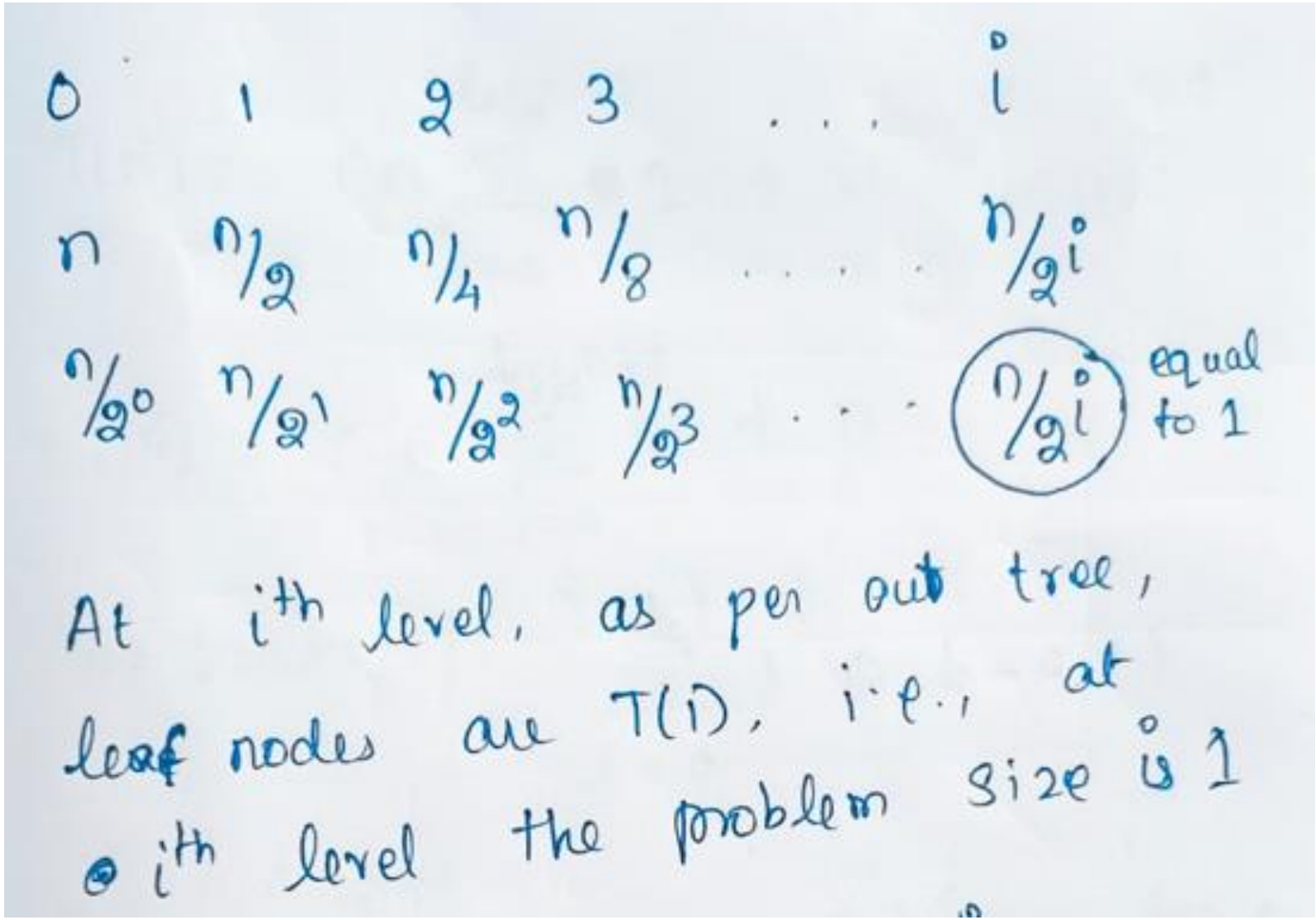
$T(n)$ = Sum of Complexity of all the nodes in the Tree.

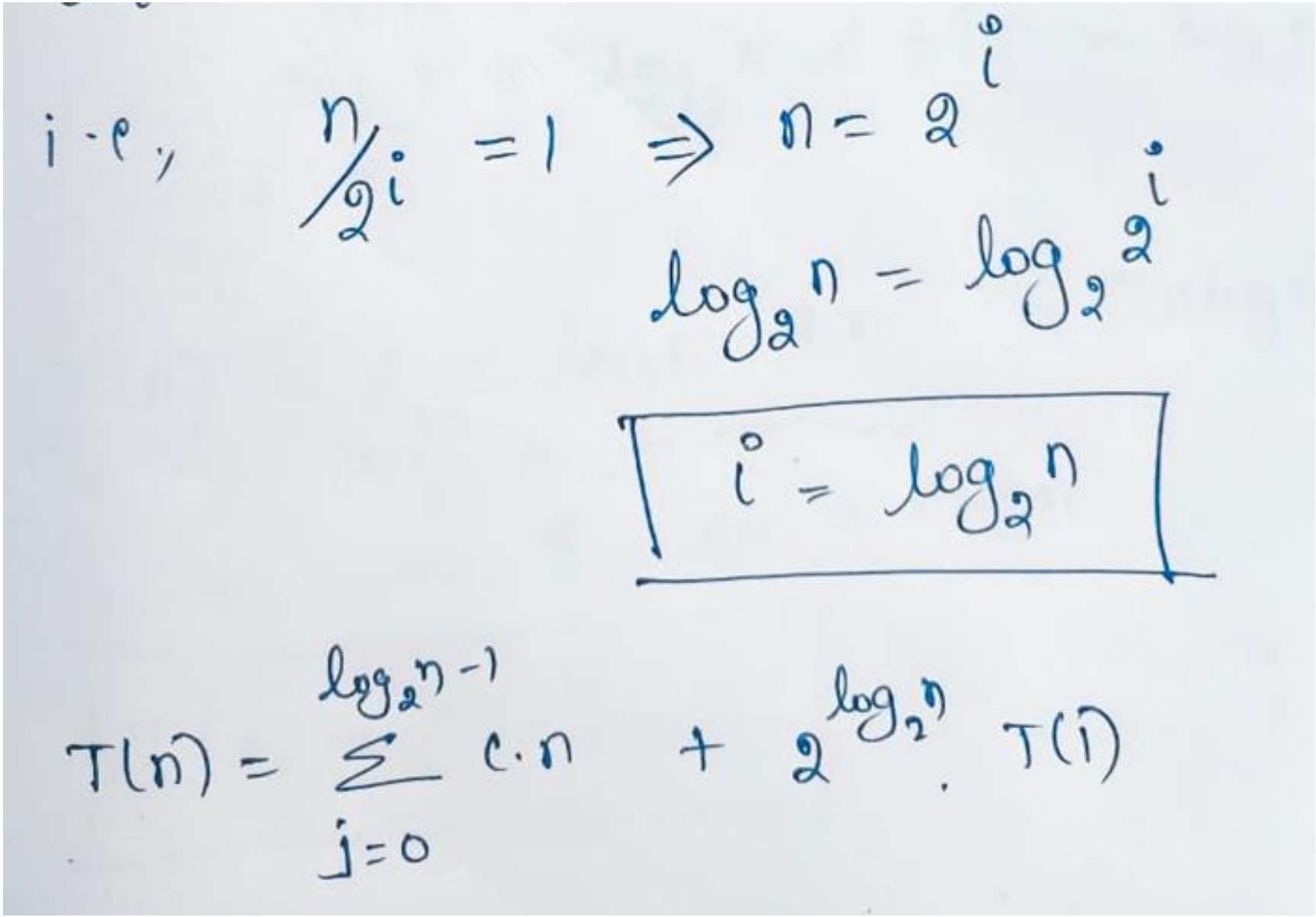
$$T(n) = \sum_{j=0}^{i-1} c \cdot n + 2^i \cdot T(1)$$

- $\sum_{j=0}^{i-1} c \cdot n$: Sum upto $i-1$ levels
- $2^i \cdot T(1)$: Sum at level i

Now need to find i value in terms of n.

For this, we need to analyze how the problem is divided in each level, i.e., what is the problem size in each level.


0 1 2 3 ... i
n n/2 n/4 n/8 ... n/2^i
n/2^0 n/2^1 n/2^2 n/2^3 ... n/2^i
equal to 1
At ith level, as per out tree,
leaf nodes are T(i), i.e., at
ith level the problem size is 1


i.e., $\frac{n}{2^i} = 1 \Rightarrow n = 2^i$
$\log_2 n = \log_2 2^i$
$i = \log_2 n$
$T(n) = \sum_{j=0}^{\log_2 n - 1} c \cdot n + 2^{\log_2 n} \cdot T(1)$

$$T(n) = c \cdot n \sum_{j=0}^{\log_2 n - 1} 1 + n^{\log_2 2} \cdot T(1) \rightarrow 1$$

$$T(n) = c \cdot n \cdot \sum_{j=0}^{\log_2 n - 1} 1 + n$$

We know that, $\sum_{l=a}^{b} 1 \Rightarrow b - a + 1$

$$\sum_{j=0}^{\log_2 n - 1} 1 = \log_2 n - 0 + 1 = \log_2 n.$$

$T(n) = \underbrace{c \cdot n \cdot \log n + n}$ $\in O(n \log n)$

Big one is $n \log n$

$$\boxed{T(n) = O(n \log_2 n)}$$

(2) $$T(n) = \begin{cases} 3 \cdot T(n/4) + n, & \text{if } n > 1 \\ O(1), & \text{if } n = 1 \end{cases}$$

No. of Sub problems $= 3$

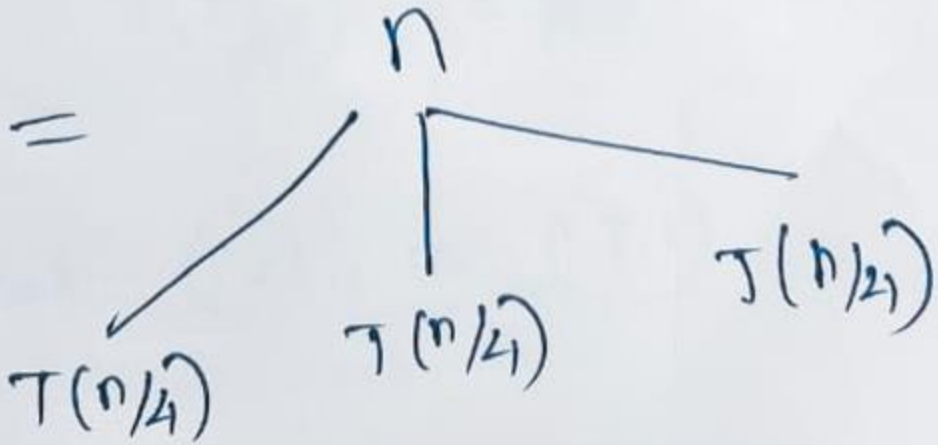
Sub problem size $= n/4$

Complexity for divide & conquer $= n$

$$T(n) = 3 \cdot T(n/4) + n \quad \rightarrow (1)$$

Corresponding Tree:

$T(n) =$



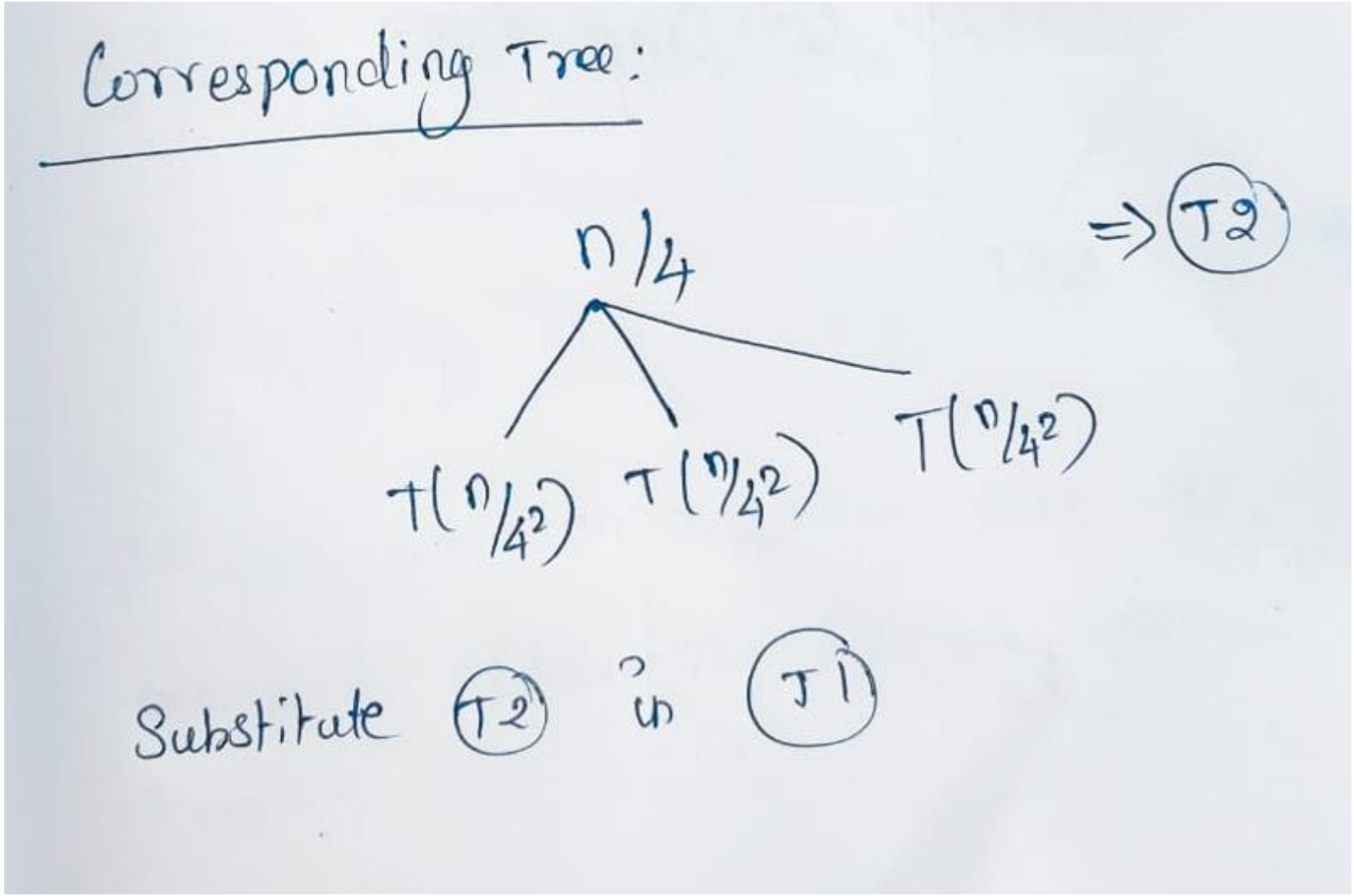
$\rightarrow (1)$

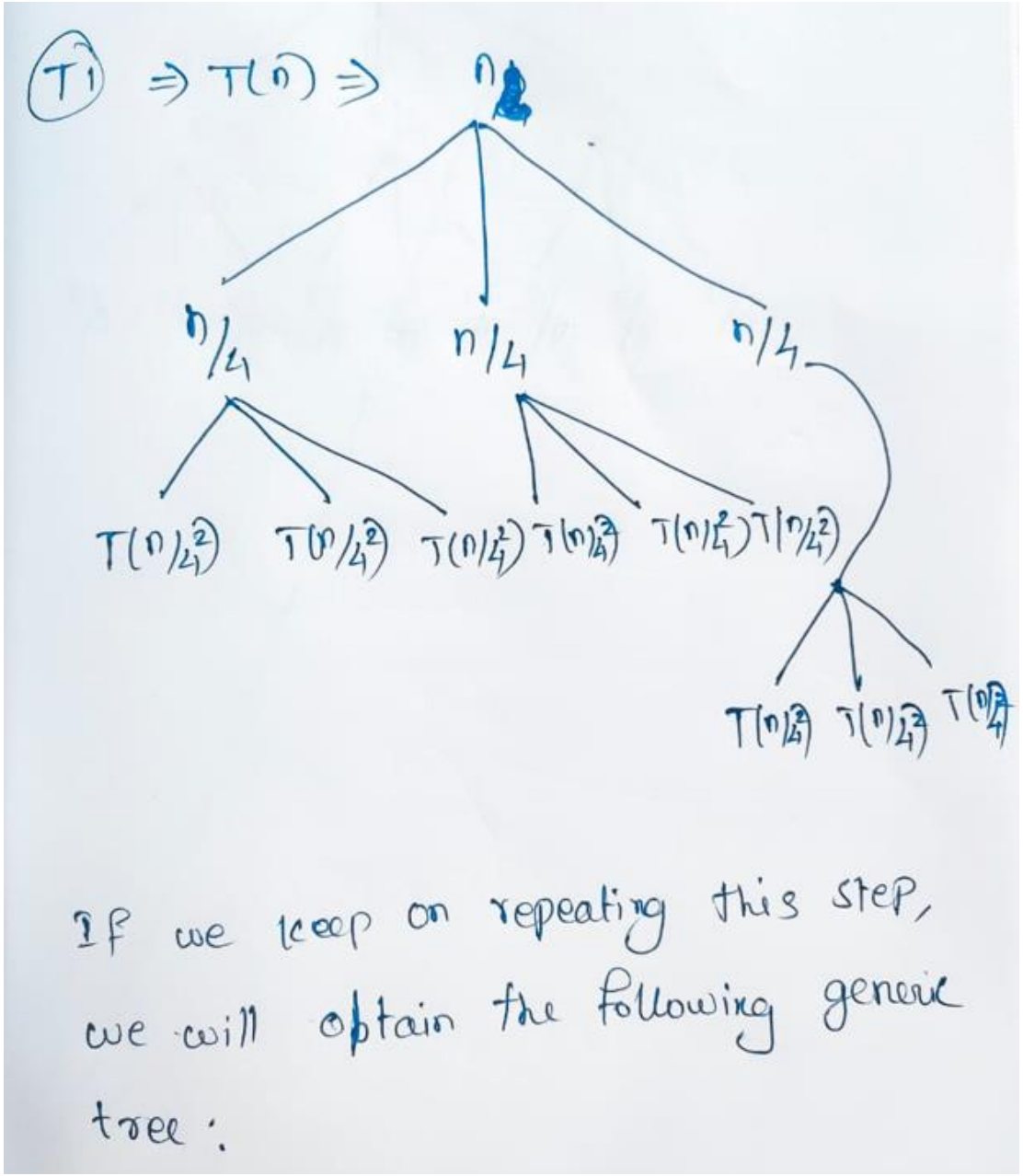
Need to find $T(n/4)$:

$n = n/4$ in (1):

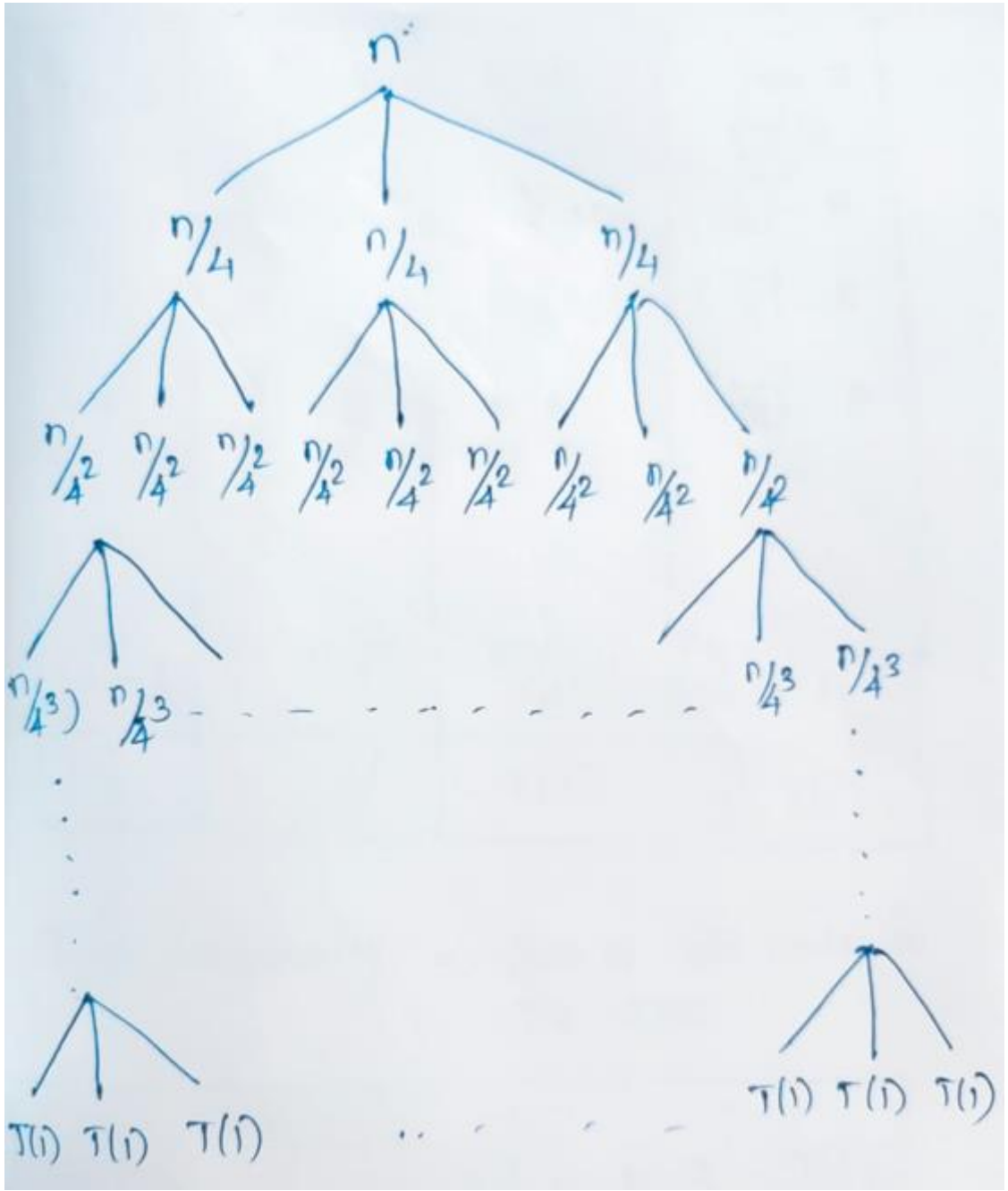
$$T(n/4) = 3 \cdot T\left(\frac{n/4}{4}\right) + n/4$$

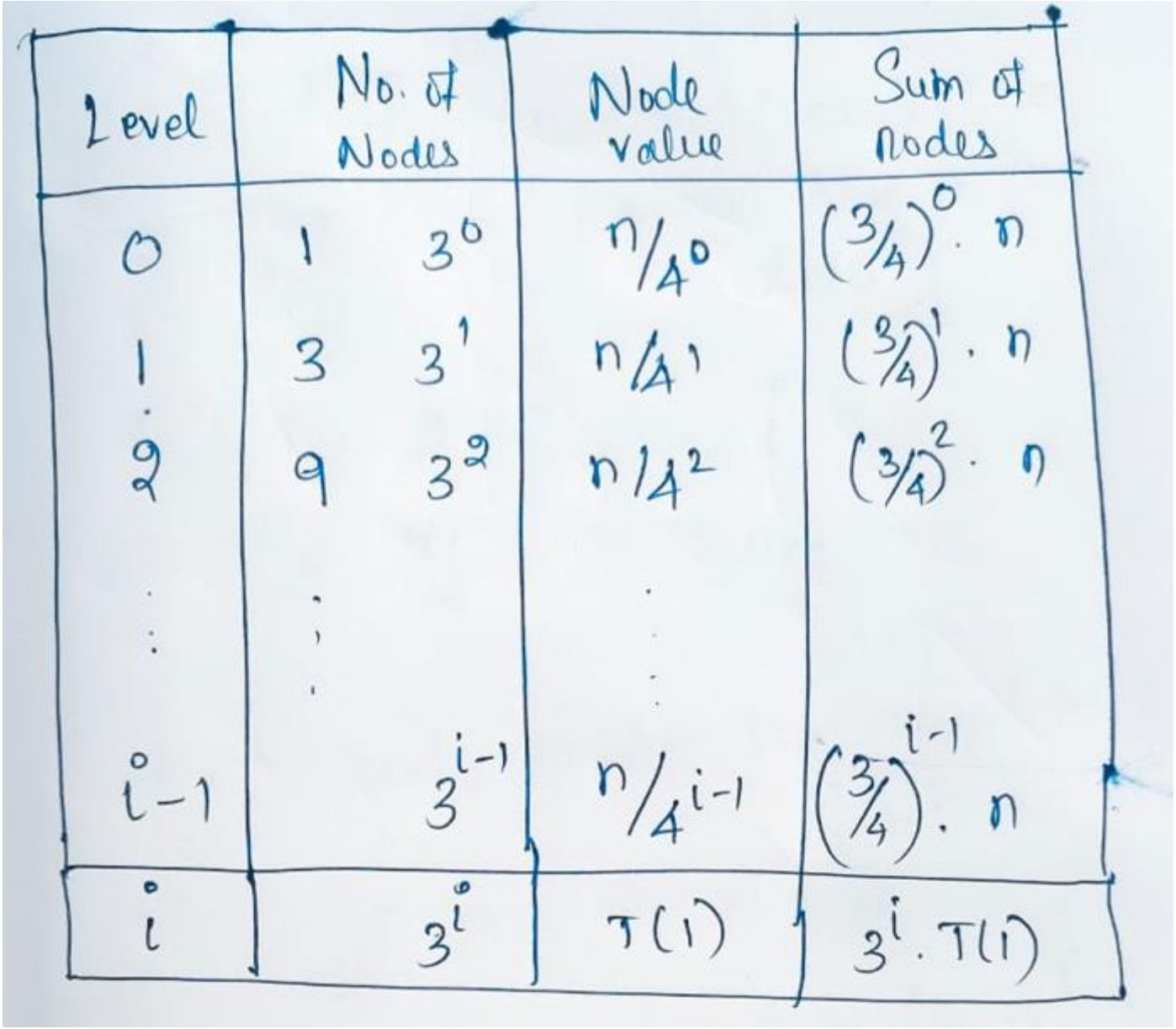
$$\boxed{T(n/4) = 3 \cdot T\left(n/4^2\right) + n/4} \rightarrow (2)$$


Corresponding Tree:
n/4
T(n/4²)
T(n/4²)
T(n/4²)
⇒ T2
Substitute T2 in T1



If we keep on repeating this step, we will obtain the following generic tree :


n
n/4
n/4
n/4
n/4²
n/4²
n/4²
n/4²
n/4²
n/4²
n/4²
n/4²
n/4²
n/4³
n/4³
n/4³
n/4³
T(1)
T(1)
T(1)
T(1)
T(1)
T(1)



Level	No. of Nodes		Node value	Sum of nodes
0	1	3^0	$n/4^0$	$(3/4)^0 \cdot n$
1	3	3^1	$n/4^1$	$(3/4)^1 \cdot n$
2	9	3^2	$n/4^2$	$(3/4)^2 \cdot n$
⋮	⋮		⋮	
$i-1$		3^{i-1}	$n/4^{i-1}$	$(3/4)^{i-1} \cdot n$
i		3^i	$T(1)$	$3^i \cdot T(1)$

Time Complexity => Sum of all nodes in the tree.

$$T(n) = \sum_{j=0}^{i-1} \left(\frac{3}{4}\right)^j \cdot n + 3^i \cdot T(1)$$

To Find i: (need to consider the problem size in each level)

0	1	2	....	i
n	$n/4$	$n/4^2$	..	$n/4^i$

At ith level, problem size equal to 1

So, $\frac{n}{4^i} = 1 \Rightarrow n = 4^i$

$\log_4 n = \log_4 4^i$

$$\boxed{i = \log_4 n}$$

$$T(n) = \sum_{j=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^j \cdot n + 3^{\log_4 n} \cdot T(1)$$

$$= n \cdot \sum_{j=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^j + n^{\log_4 3} \cdot (1)$$

For large n, $\sum_{i=0}^{\alpha} x^i = \frac{1}{1-x}$.

$$T(n) = n \cdot \left(\frac{1}{1-3/4}\right) + n^{0.81}$$

$$T(n) = \underbrace{4 \cdot n + n^{0.81}}_{\text{Big one } n^1} \in O(n)$$

$$\boxed{T(n) = O(n)}$$

Problems:

(3) $$T(n) = \begin{cases} 2 \cdot T(n/5) + n, & \text{if } n>1 \\ \theta(1), & \text{if } n=1 \end{cases}$$

(4) $$T(n) = \begin{cases} T(n-1)+T(n-2)+1, & \text{if } n>2 \\ \theta(1), & \text{if } n=1 \text{ or } n=2 \end{cases}$$

(5) $$T(n) = \begin{cases} T(n/3)+T(2n/3)+n, & \text{if } n>1 \\ \theta(1), & \text{if } n=1 \end{cases}$$

Solving Recurrences – Using Master Theorem

MASTER THEOREM:

(for Solving Reccurences)

Let $a \geq 1$ and $b \geq 1$. Let $f(n)$ be a function. Let $T(n)$ be defined on non-negative integers by the reccurence.

$$T(n) = a \cdot T(n/b) + f(n)$$

$T(n)$ has the following asymptotic bounds.

Case - 1:

If $f(n) = O(n^{(\log_b a) - \varepsilon})$ for some constant $\varepsilon > 0$, then the solution

$$\boxed{T(n) = \theta(n^{\log_b a})}$$

Case - 2:

If $f(n) = \Theta(n^{\log_b a})$, then the solution

$$\boxed{T(n) = \Theta(n^{\log_b a} \cdot \log_b n)}$$

Case - 3:

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for

Some Constant $\epsilon > 0$,

AND If $a \cdot f(n/b) \leq c \cdot f(n)$ for

Some Constant $c < 1$ and all

Sufficient large 'n'.

If both conditions are true,

then the Solution

$$T(n) = \Theta(f(n))$$

Problems:

(1) $T(n) = 2 \cdot T(n/4) + 1$

Step-1: Identify $a, b, f(n)$

$a = 2 \quad b = 4 \quad f(n) = 1$

Step-2: Find $n^{\log_b a}$

$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

Step-3: Represent $f(n)$ as n to the power of Something.

$$f(n) = 1 = n^0$$

$$= n^{0.5-0.5}$$

$$f(n) = n^{\log_4 2 - 0.5} \Rightarrow n^{\log_b a - \epsilon}$$

$$f(n) = \Theta\left(n^{\log_4 2 - 0.5}\right) \Rightarrow O\left(n^{\log_b a - \epsilon}\right)$$

with $\epsilon = 0.5 > 0$

This belongs to Case-1: So the Solution for case-1 is $\theta(n^{\log_b a})$

So the Solution for this reccurence is:

$$T(n) = \theta(n^{\log_b a})$$

$$= \theta(n^{\log_4 2}) = \theta(n^{0.5})$$

$$\boxed{T(n) = \theta(\sqrt{n})}$$

(2) $T(n) = 2 \cdot T(n/4) + \sqrt{n}$

Step-1: Identify a, b, f(n)

$a = 2$ $b = 4$ $f(n) = \sqrt{n}$

Step-2: Find $n^{\log_b a}$

$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

Step-3: Find the case by checking f(n)

$$f(n) = \sqrt{n} = n^{0.5}$$

$$= n^{\log_4 2} = \Theta(n^{\log_4 2})$$

$$\boxed{f(n) = \Theta(n^{\log_4 2})} \Rightarrow \Theta(n^{\log_b a})$$

This belongs to case-2:

Case-2 Solution: $T(n) = \theta(n^{\log_b a} \cdot \log n)$

$$T(n) = \theta(n^{\log_4 2} \cdot \log n)$$

$$T(n) = \theta(\sqrt{n} \cdot \log n)$$

(3) $T(n) = 2.T(n/4) + n$

Step1: $a = 2 \quad b = 4 \quad f(n) = n$

Step2: $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

Step3: $f(n) = n = n^1 = n^{0.5+0.5}$

$f(n) = n^{\log_4 2 + 0.5}$

$$f(n) = \Omega\left(n^{\log_4 2 + 0.5}\right)$$

$$\hookrightarrow f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$$

with $\varepsilon = 0.5 > 0$.

So this may belongs to case 3:

But we need to check one more condition for case 3.

Need to verify

$a \cdot f(n/b) \leq c \cdot f(n)$ for $c < 1$

$2 \cdot f(n/4) \leq c \cdot f(n)$

↳ need to find.

if $f(n) = n$, then $f(n/4) = n/4$

$2 \cdot n/4 \leq c \cdot F(n)$

$2 \cdot n/4_2 \leq c \cdot n$

$\left(\frac{1}{2}\right) n \leq c \cdot n$

At worst case : $\frac{1}{2} n = c \cdot n$

i.e., $\boxed{c = 0.5} < 1$

So 2nd condition also true.

So, this recurrence belongs to case 3:

Case 3 Solution is: $T(n) = \Theta(f(n))$

$$\boxed{T(n) = \Theta(n)}$$

(4) $T(n) = 2 \cdot T(n/4) + n^2$

$a = 2 \quad b = 4 \quad f(n) = n^2$

$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

$f(n) = n^2 = n^{0.5+1.5} = \Omega(n^{\log_4 2 + 1.5})$

$$\boxed{f(n) = \Omega(n^{\log_4 2 + 1.5})}$$

$$\hookrightarrow f(n) = \Omega(n^{\log_b a + 1.5})$$

with $\epsilon = 1.5 > 0$.

May belongs to case 3:

Need to verify:

$a \cdot f(n/b) \leq c \cdot f(n)$ for $c < 1$

$2 \cdot f(n/4) \leq c \cdot f(n)$

if $f(n) = n^2$ then $f(n/4) = n^2/16$

$2 \cdot n^2/16 \leq c \cdot n^2$

$(1/8) \cdot n^2 \leq c \cdot n^2$.

$\boxed{c = 1/8} < 1 \Rightarrow$ True.

So belongs to case 3:

Case 3 Solution: $T(n) = \Theta(f(n))$

So: $\boxed{T(n) = \Theta(n^2)}$

(5) $T(n) = 2 \cdot T(n/2) + n$

$a = 2 \quad b = 2 \quad f(n) = n$

$n^{\log_b a} = n^{\log_2 2} = n^1$

$f(n) = n = n^1 = \theta(n^1) = \theta(n^{\log_2 2})$

$F(n) = \theta(n^{\log_b a})$.

So this belongs to case 2:

Case 2 Solution: $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

$T(n) = \Theta(n^{\log_2 2} \cdot \log n)$

$$\boxed{T(n) = \Theta(n \cdot \log n)}$$

Additional Problems:

(6) $T(n) = 9 \cdot T(n/3) + n$

(7) $T(n) = 0 \cdot T(2n/3) + 1$

(8) $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$

(9) $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$

(16) $T(n) = 3 \cdot T(n/4) + n \log_2 n$

Solve using Recursion Tree Method:

(5) $$T(n) = \begin{cases} T(n/3) + T(2n/3) + n, & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

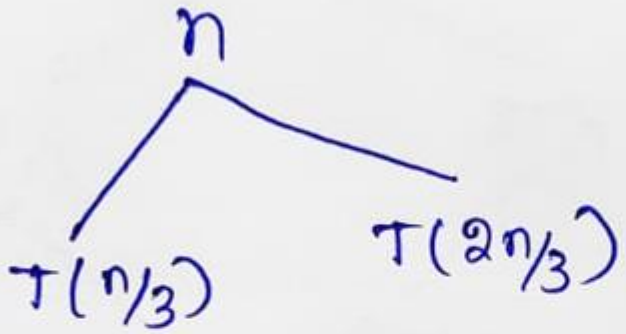
No. of Sub problems = 2

Sub problems Size = $n/3$ & $2n/3$

Complexity for divide & Combine = n

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

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

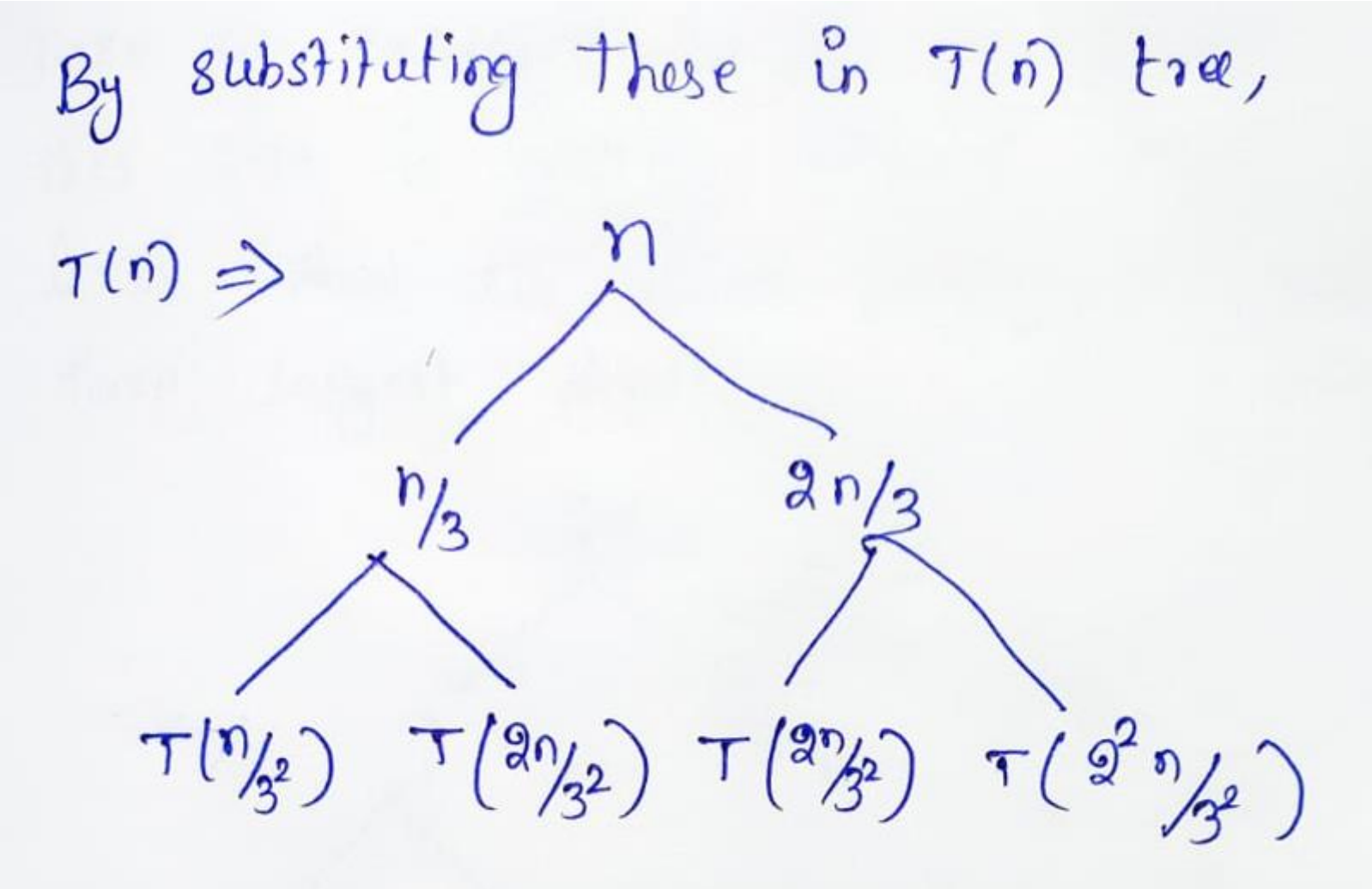


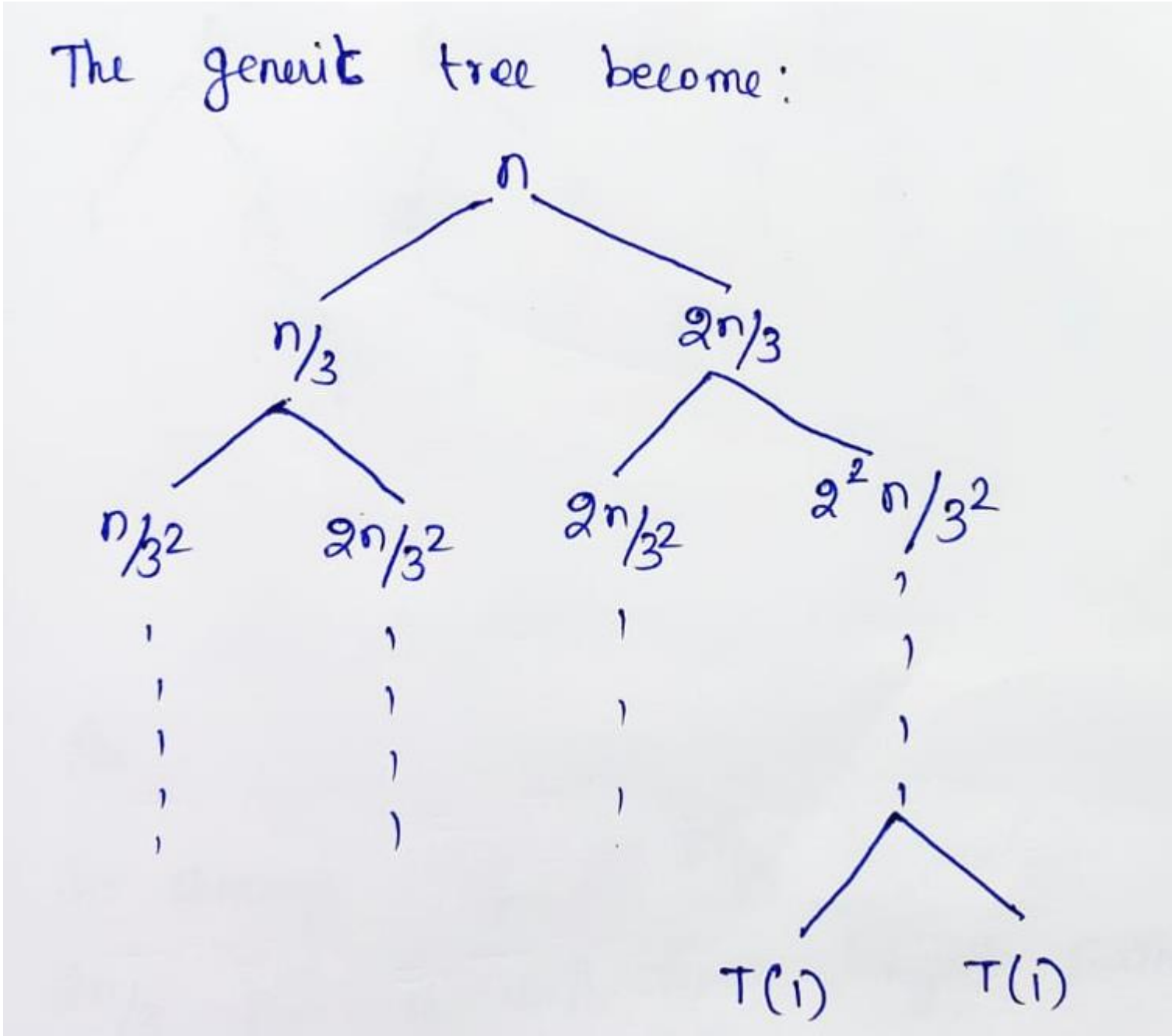
Need to find $T(n/3)$ & $T(2n/3)$ from $T(n)$

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

$$T(2n/3) = T(2n/3^2) + T(2^2 \cdot n/3^2) + 2n/3$$

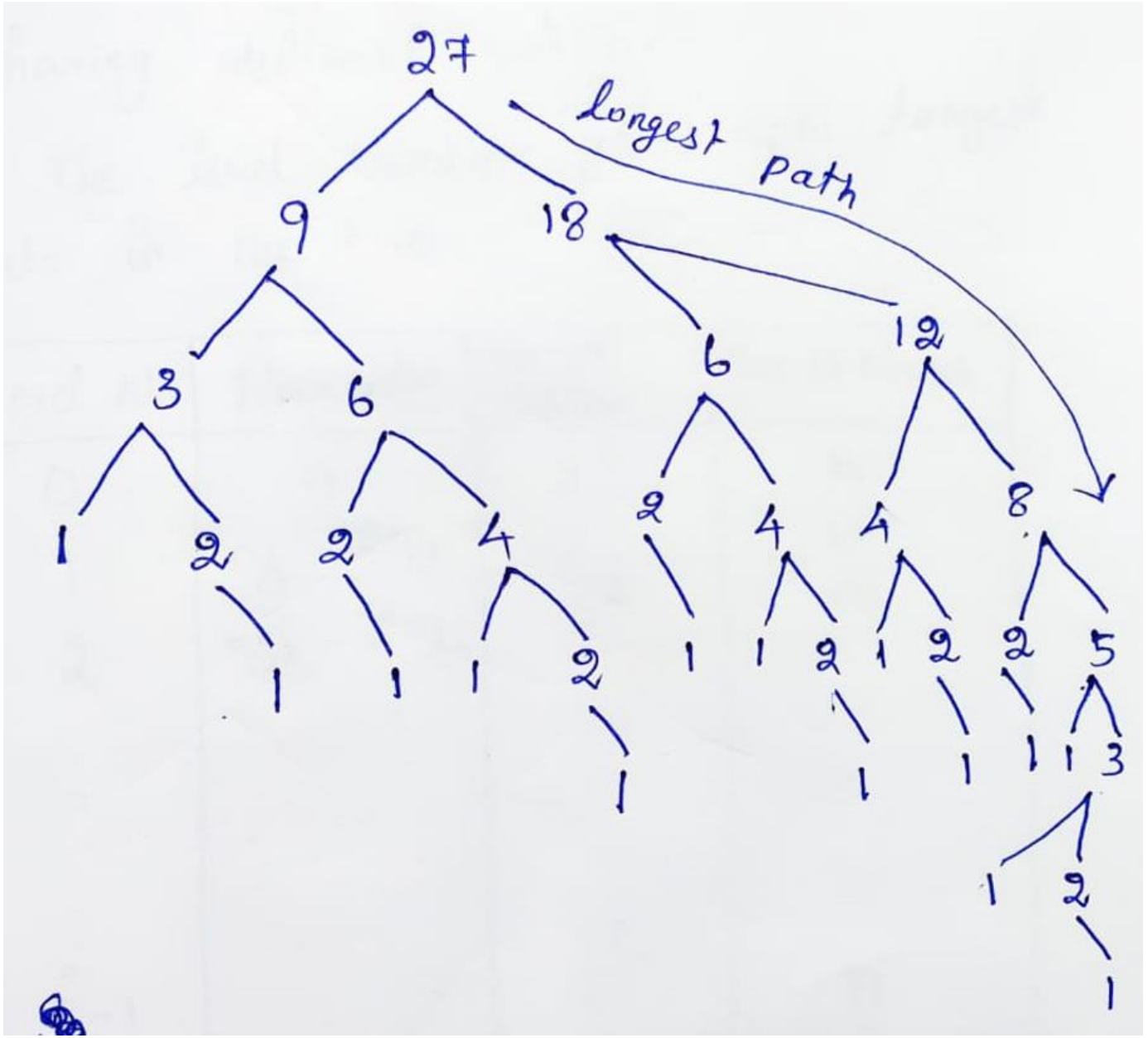
By substituting these in T(n) tree,




The genrit tree become:
n
n/3
2n/3
n/3^2
2n/3^2
2n/3^2
2^2 n/3^2
T(1)
T(1)

Take an example $n=27$, See how this size is getting reduced in each level. And see which branch will have longest depth.

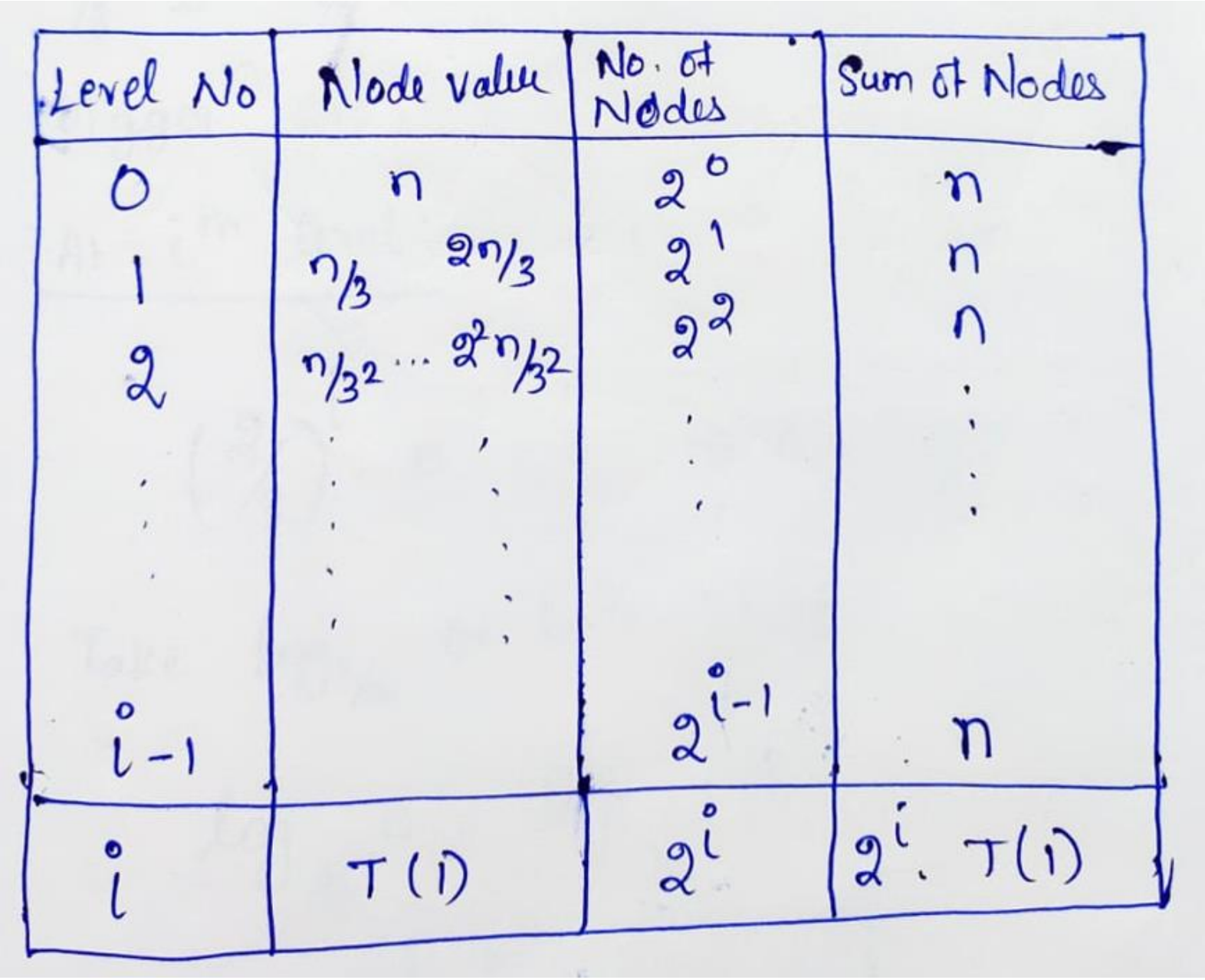
So among $n/3$ & $2n/3$ branches, $2n/3$ branch will have longest path.


27
longest Path
9
18
3
6
6
12
1
2
2
4
2
4
4
8
1
1
1
2
1
1
1
2
1
2
2
5
1
1
1
1
1
3
1
2
1

We need to consider the longest path while finding sum of the nodes.

While drawing table, Node value cannot be written, since, the 2 branches are having different values.

The level number goes upto longest path in the tree.



Level No	Node value	No. of Nodes	Sum of Nodes
0	n	2^0	n
1	$n/3$ $\quad$ $2n/3$	2^1	n
2	$n/3^2$ ... $2^2 n/3^2$	2^2	n
.	.	.	.
.	.	.	.
.	.	.	.
$i-1$		2^{i-1}	n
i	$T(1)$	2^i	$2^i . T(1)$

To Find i:

0	1	2	3	....	i^{th}
n	$2n/3$	$2^2 n/3^2$	$2^3 n/3^3$	....	$2^i \cdot n/3^i$

When the size n is divided into $n/3$ & $2n/3$, we need to consider the bigger one,

At i^{th} level: Size $= 2^i \cdot n/3^i = 1$

$$\left(\frac{2}{3}\right)^i \cdot n = 1 \Rightarrow n = \left(\frac{3}{2}\right)^i$$

Take $\log_{3/2}$ on both sides:

$$\log_{3/2} n = \log_{3/2} \left(\frac{3}{2}\right)^i$$

$$\boxed{i = \log_{3/2} n}$$

Now need to find the Sum:

$$T(n) \leq \left[\text{Sum in } 0 + \text{Sum in } 1 + \cdots \text{Sum in } i-1 \right] + \text{Sum in } i$$

$$T(n) \leq \left[n + n + \cdots + n \right] + 2^i . T(1)$$

$$T(n) \leq \sum_{j=0}^{i-1} n + 2^i . T(1)$$

$$T(n) \leq n . \sum_{j=0}^{i-1} 1 + 2^i . T(1) \quad \Big| \sum_{j=0}^{n} 1 = n+1$$

$$T(n) \leq n.(i-1+1) + 2^i.T(1)$$

$$T(n) \leq n.i + 2^i.T(1)$$

$i = \log_{3/2} n$

$T(1) = 1$

$$T(n) \leq n.\log_{3/2} n + 2^{\log_{3/2} n}$$

$$T(n) \leq \underbrace{n.\log_{3/2} n + n^{\log_{3/2} 2}}_{\max \text{ is } n^{\log_{3/2} 2}}$$

Solution:

$$T(n) = O(n^{\log_{3/2} 2})$$

Solving Recurrences – Problems in All Methods

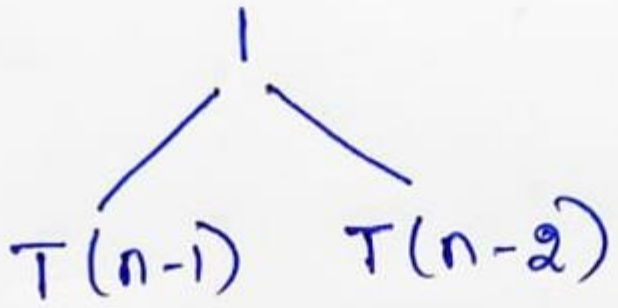
Using Recursion Tree Method:

(b) $$T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & n > 2 \\ \theta(1) & n = 1 \text{ or } n = 2 \end{cases}$$

Recursive case:

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

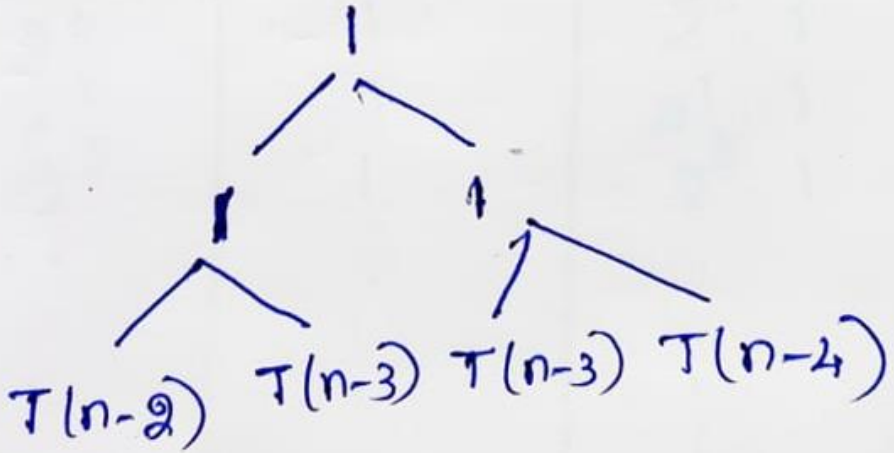
Tree:



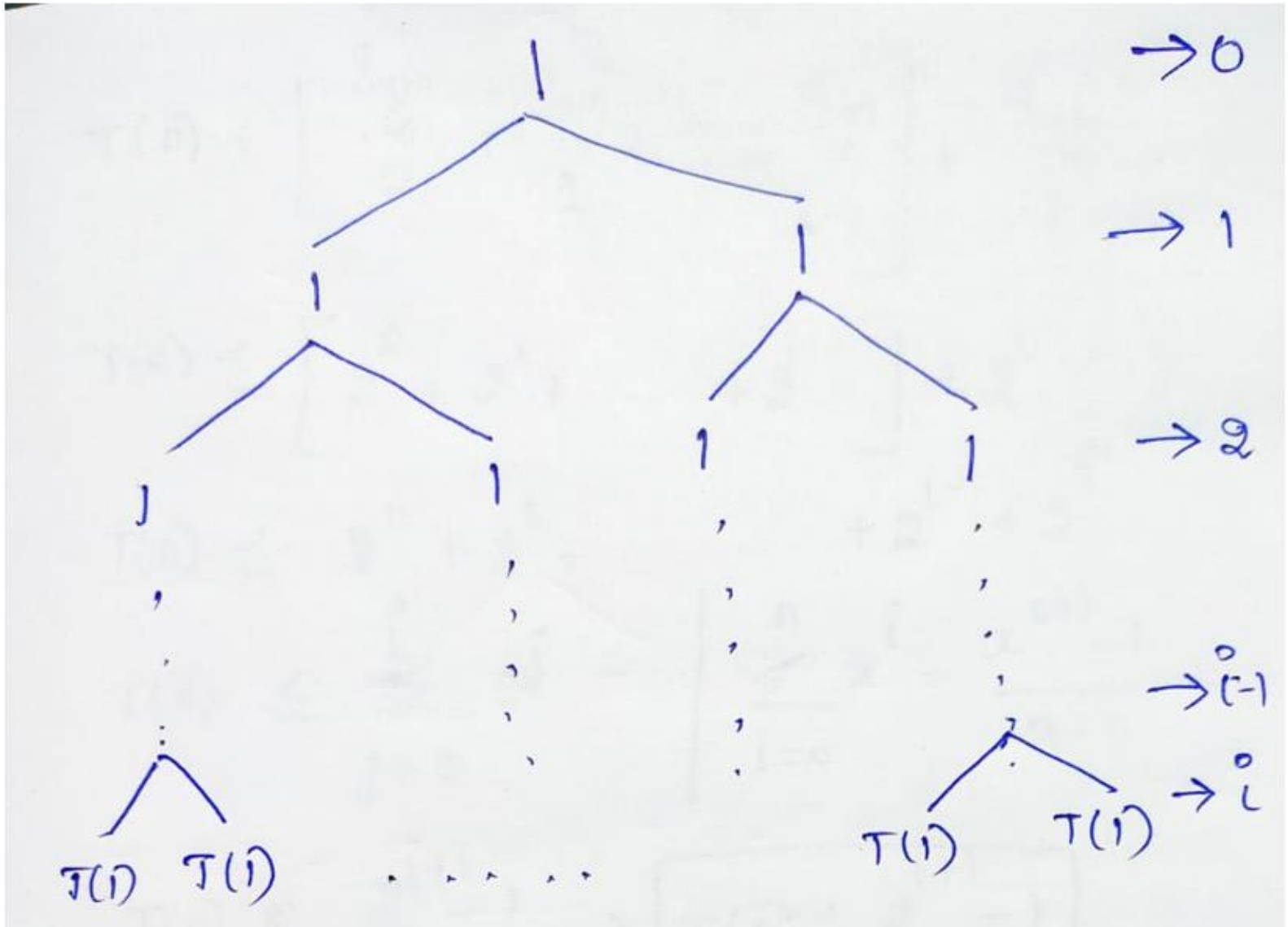
$T(n-1) = T(n-2) + T(n-3) + 1$

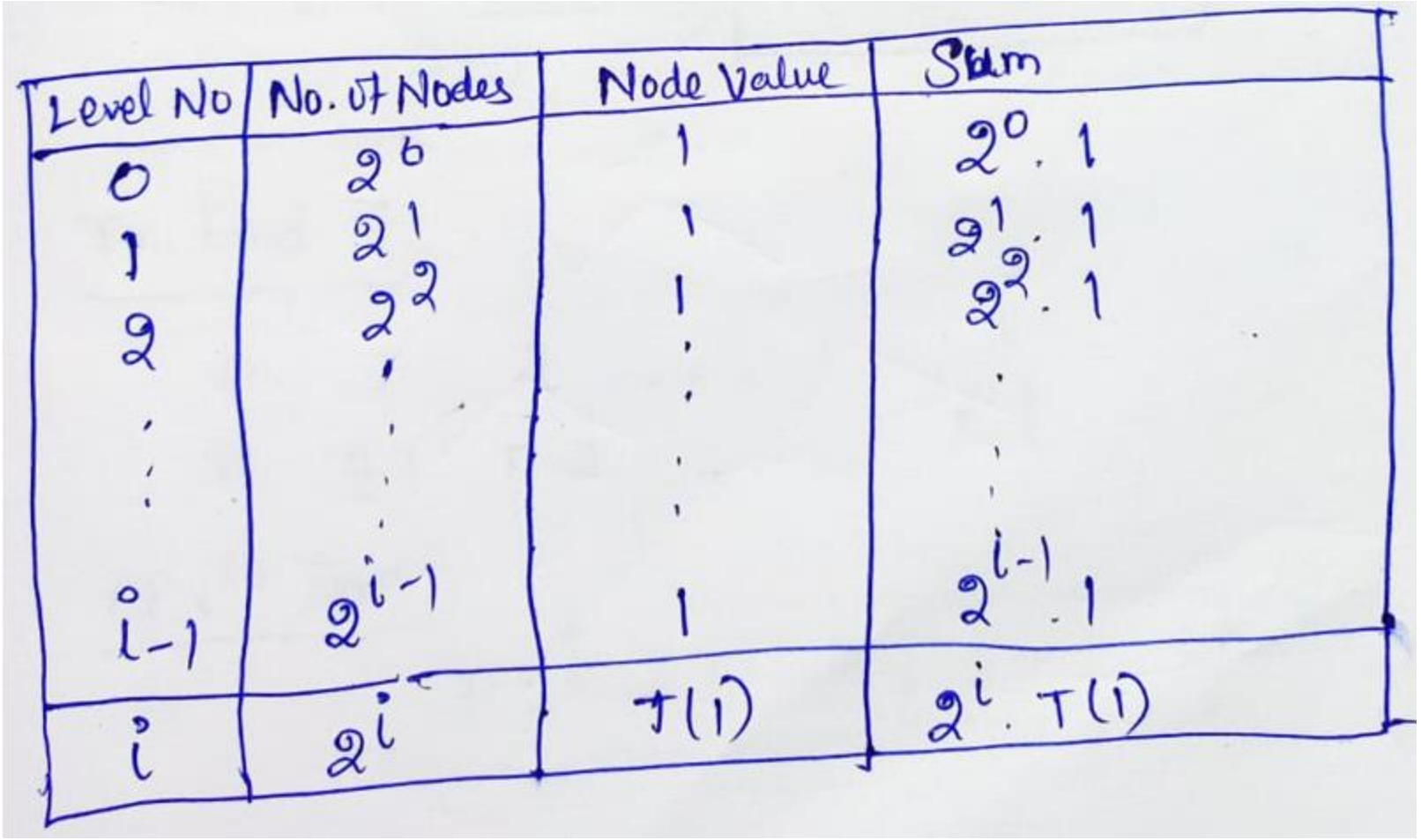
$T(n-2) = T(n-3) + T(n-4) + 1$

From $T(n)$:
$n = n-1$ &
$n = n-2$



By extending the levels of Tree:


→ 0
→ 1
→ 2
→ i-1
→ i
T(1)
T(1)
T(1)
T(1)



Level No	No. of Nodes	Node Value	Sum
0	2^0	1	$2^0 \cdot 1$
1	2^1	1	$2^1 \cdot 1$
2	2^2	1	$2^2 \cdot 1$
.	.	.	.
.	.	.	.
.	.	.	.
i-1	2^{i-1}	1	$2^{i-1} \cdot 1$
i	2^i	T(1)	$2^i \cdot T(1)$

$$T(n) \leq \left[\underset{0}{\overset{n}{Sum}} + \underset{1}{\overset{n}{Sum}} + \ldots + \underset{i-1}{\overset{n}{Sum}} \right] + \underset{i}{\overset{n}{Sum}}$$

$$T(n) \leq \left[2^0 + 2^1 + \ldots + 2^{i-1} \right] + 2^i . T(1)$$

$$T(n) \leq 2^0 + 2^1 + \ldots \ldots + 2^{i-1} + 2^i$$

$$T(n) \leq \sum_{j=0}^{i} 2^j \quad \Big| \quad \sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

$$T(n) \leq \frac{2^{i+1}-1}{2-1} \Rightarrow \boxed{T(n) \leq 2^{i+1}-1}$$

To Find i:

0	1	2	...	i
n	$n-1$	$n-2$	...	$n-i$

At i^{th} level:

$$n - i = 1 \Rightarrow \boxed{i = n-1}$$

$$T(n) \leq 2^{n-1+1} - 1 \Rightarrow T(n) \leq 2^{n} - 1 \in O(2^n)$$

$$\boxed{T(n) = O(2^n)}$$

↓ Ignore -ve Term.

Solve using Iterative Method:

④ $T(n) = \begin{cases} T(n-1)+n & , \text{if } n>1 \\ \theta(1) & , \text{if } n=1 \end{cases}$

Recursive case:

$$T(n) = T(n-1)+n \quad \rightarrow (1)$$

$n = n-1$ in (1)

$$T(n-1) = T(n-2)+n-1 \quad \rightarrow (2)$$

(2) in (1)

$$T(n) = T(n-2)+(n-1)+n \quad \rightarrow (3)$$

$n = n-2$ in (1)

$$T(n-2) = T(n-3)+(n-3) \quad \rightarrow (4)$$

④ in ③

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad \rightarrow ⑤$$

From ①, ③, ⑤, In general,

$$T(n) = T(n-i) + (n-(i-1)) + (n-(i-2)) + \ldots + (n-(i-i))$$

At i^{th} level:

$n - i$ = Problem Size = 1

$n - i = 1 \Rightarrow \boxed{i = n-1}$

$$T(n) = \sum_{j=0}^{i-1} (n-j) + T(n-i)$$

$$T(n) = \sum_{j=0}^{n-1-1} (n-j) + T(n-(n-1))$$

$$T(n) = \left[(n-0) + (n-1) + (n-2) \cdots + n-(n-2) \right] + T(n-(n-1))$$

$$T(n) = [n + (n-1) + (n-2) + \cdots + 2] + T(1)$$

$$= [n + (n-1) + (n-2) + \cdots + 2] + 1$$

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

Its a arithmetic Series

$$T(n) = \frac{n(n+1)}{2} \in O(n^2)$$

$$\boxed{T(n) = O(n^2)}$$

Using Master Theorem:

(6) $T(n) = 9 \cdot T(n/3) + n$

$a = 9 \quad b = 3 \quad f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 3^2} = n^2$

$$\boxed{n^{\log_b a} = n^2}$$

$$f(n) = n^1 = n^{2-1} = n^{\log_3 9 - 1}$$

$$f(n) = O\left(n^{\log_3 9 - 1}\right) \Rightarrow \underline{\text{Case 1}}$$

with

$\varepsilon = 1 > 0$

Case 1 Solution:

$$T(n) = \theta\left(n^{\log_b a}\right)$$

$$T(n) = \theta\left(n^{\log_3 9}\right)$$

$$\boxed{T(n) = \theta(n^2)}$$

(7) $T(n) = T(2n/3) + 1$

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

$a = 1 \quad b = 3/2 \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 \Rightarrow \boxed{n^{\log_b a} = n^0}$

$$f(n) = 1 = n^0 = \theta(n^0)$$

$$f(n) = \theta(n^{\log_{3/2} 1}) \Rightarrow \text{Case 2}$$

Case 2 Solution:

$$T(n) = \theta(n^{\log_b a} \cdot \log n)$$

$$T(n) = \theta(n^{\log_{3/2} 1} \cdot \log n)$$

$$T(n) = \theta(1 \cdot \log n)$$

$$\boxed{T(n) = \theta(\log n)}$$

(8) $T(n) = 8 \cdot T(n/2) + \theta(n^2)$

$a = 8 \quad b = 2 \quad f(n) = \theta(n^2)$

$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \boxed{n^{\log_b a} = n^3}$

$f(n) = \theta(n^2) = \theta(n^{3-1}) = \theta(n^{\log_2 8 - 1})$

$\Rightarrow$ Case 1 with $\varepsilon = 1 > 0$

Case 1 Solution: $T(n) = \theta(n^{\log_b a})$

$T(n) = \theta(n^{\log_2 8}) \Rightarrow \boxed{T(n) = \theta(n^3)}$

(9) $T(n) = 7 \cdot T(n/2) + \theta(n^2)$

$a = 7 \quad b = 2 \quad f(n) = \theta(n^2)$

$n^{\log_b a} = n^{\log_2 7} = n^{2.81} \Rightarrow \boxed{n^{\log_b a} = n^{2.81}}$

$f(n) = \theta(n^2) = \theta(n^{2.81-0.81})$

$= \theta(n^{\log_2 7 - 0.81})$

$\Rightarrow$ case 1 with $\varepsilon = 0.81 > 0$

Case 1 Solution: $T(n) = \theta(n^{\log_b a})$

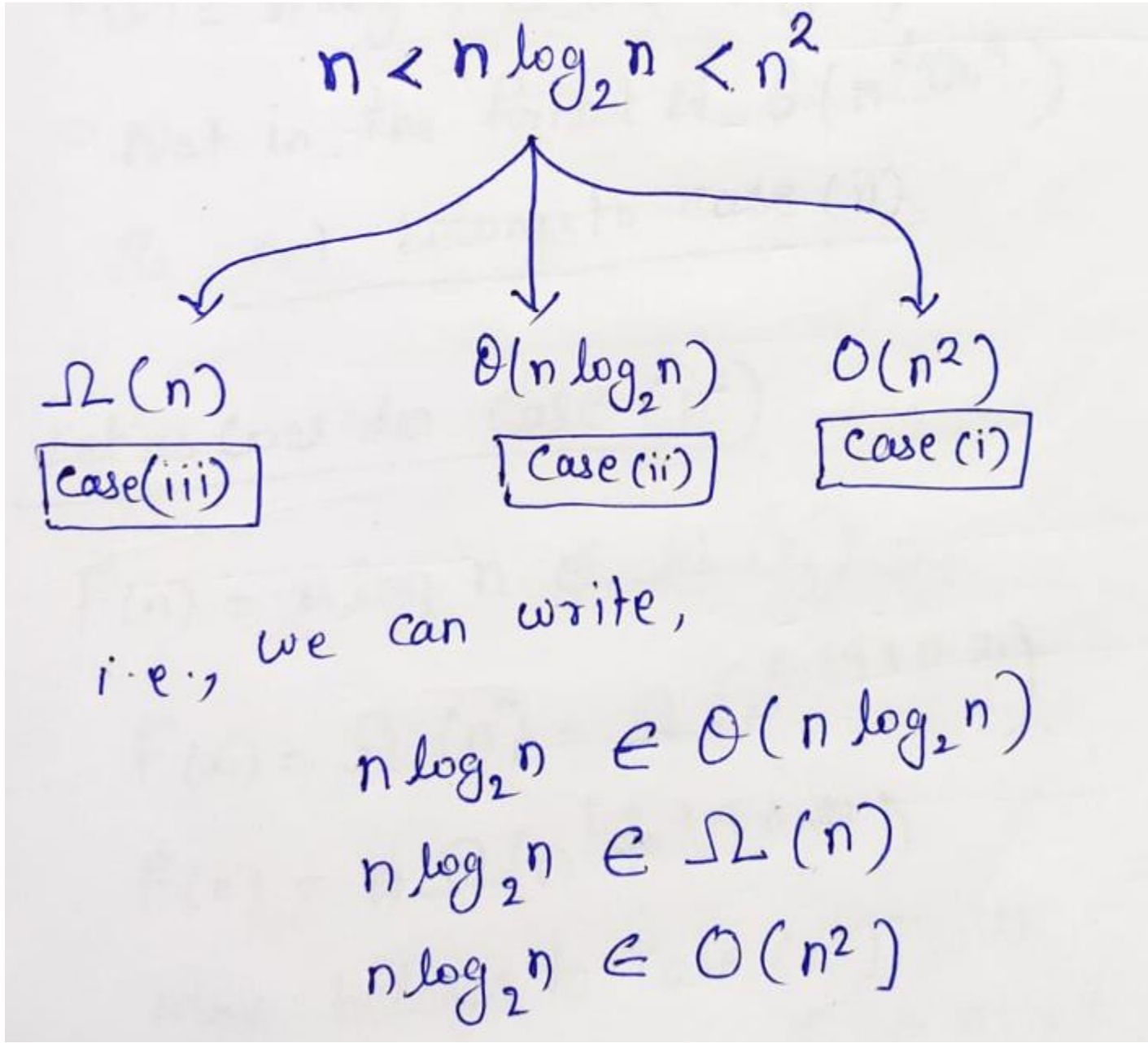
$T(n) = \theta(n^{\log_2 7}) \Rightarrow \boxed{T(n) = \theta(n^{2.81})}$

(10) $T(n) = 3 \cdot T(n/4) + n \log_2 n$

$a = 3 \quad b = 4 \quad f(n) = n \log_2 n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.79} \Rightarrow \boxed{n^{\log_b a} = n^{0.79}}$

$f(n) = n \log_2 n$


$n < n\log_2 n < n^2$
$\Omega(n)$
$\theta(n\log_2 n)$
$O(n^2)$
Case(iii)
Case (ii)
Case (i)
i.e., we can write,
$n\log_2 n \in \theta(n\log_2 n)$
$n\log_2 n \in \Omega(n)$
$n\log_2 n \in O(n^2)$

Let us consider case 1:

$f(n) = n \log n \in O(n^2)$

$f(n) = O(n^{0.79+1.21}) \Rightarrow$ Not belongs to case (i),

Because, case1 format is $O(n^{\log_b a - \epsilon})$

Let us Consider Case(ii)

$f(n) = n \log n \in \Theta(n \log n)$

Not in the format of $\Theta(n^{\log_b a})$

So not belongs to case (ii)

Let us Consider Case (iii)

$f(n) = n \log n \in \Omega(n)$

$f(n) = \Omega(n) = \Omega\left(n^{0.79+0.21}\right)$.

$f(n) = \Omega\left(n^{\log_b a + 0.21}\right)$

May belongs to case (iii) with $\epsilon = 0.21 > 0$

Now, need to verify another condition for case (iii):

$a \cdot f(n/b) \leq c \cdot f(n)$ for $c < 1$

$3 \cdot f(n/4) \leq c \cdot n \log n$

$f(n) = n \log n \Rightarrow f(n/4) = n/4 \cdot \log n/4.$

$$3 \cdot n/4 \cdot \log n/4 \leq c \cdot n \log n$$

$$(3/4) \cdot n(\log n - \log 4) \leq c \cdot n \log n$$

$$(3/4) \cdot n \cdot \log n - 3/4 \cdot \log_2 4 \leq c \cdot n \log n$$

$$3/4\, n \cdot \log n - 3/2 \leq c \cdot n \log n$$

$-3/2$ is negative constant. So we can ignore it.

$\frac{3}{4} \cdot n \cdot \log n \leq c \cdot n \cdot \log n$

where $c = \frac{3}{4} < 1 \Rightarrow$ Condition True.

So this belongs to case (iii)

Case 3 Solution: $T(n) = \theta(F(n))$

$$T(n) = \theta(n \log_2 n)$$

PROBLEMS - MASTER THEOREM:

1. $T(n) = 3.T(n/2) + n^2$
2. $T(n) = 4.T(n/2) + n^2$
3. $T(n) = T(n/2) + 2^n$
4. $T(n) = 2^n.T(n/2) + n^n$
5. $T(n) = 16.T(n/4) + n$
6. $T(n) = 2.T(n/2) + n.\log_2 n$
7. $T(n) = 2.T(n/2) + n/\log_2 n$
8. $T(n) = 2.T(n/4) + n^{0.51}$
9. $T(n) = 0.5\,T(n/2) + 1/n$
10. $T(n) = 16.T(n/4) + n!$
11. $T(n) = \sqrt{2}.T(n/2) + \log_2 n$
12. $T(n) = 3.T(n/2) + n$
13. $T(n) = 3.T(n/3) + \sqrt{n}$
14. $T(n) = 4.T(n/2) + c.n$
15. $T(n) = 3.T(n/4) + n.\log_2 n$
16. $T(n) = 3.T(n/3) + n/2$
17. $T(n) = 6.T(n/3) + n^2.\log_2 n$
18. $T(n) = 4.T(n/2) + n/\log_2 n$
19. $T(n) = 64.T(n/8) - n^2.\log_2 n$
20. $T(n) = 7.T(n/3) + n^2$
21. $T(n) = 4.T(n/2) + \log_2 n$
22. $T(n) = T(n/2) + n\,(2 - \cos n)$
23. $T(n) = 4.T(n/4) + n^2$
24. $T(n) = 4.T(n/2) + n^2$
25. $T(n) = 8.T(n/2) + n^2$
26. $T(n) = T(n-1) + n$
27. $T(n) = 7.T(n/2) + 9/2\,n^2$

Solving Recurrences – Substitution Method

Substitution Method

- ✓ The substitution method for solving recurrences comprises two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.
- ✓ We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name "substitution method."
- ✓ This method is powerful, but we must be able to guess the form of the answer in order to apply it.
- ✓ We can use the substitution method to establish either upper or lower bounds on a recurrence.

Substitution Method - Example

Determine a tight asymptotic lower bound for the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2.$$

Let us guess that $T(n) = n^2 \lg(n)$. Then our induction hypothesis is that there exists a c and an n_0 such that

$$T(n) \geq cn^2 \lg(n) \; \forall n > n_0 \;\text{ and }\; c > 0.$$

For the base case $(n = 1)$, we have $T(1) = 1 > c1^2 \lg 1$. This is true for all $c > 0$.

Now, for the inductive step, assume the hypothesis is true for $m < n$. Then

$$T(m) \geq cm^2 \lg(m).$$

So,

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n^2 \\ &\geq 4c\frac{n^2}{4} \lg \frac{n}{2} + n^2 \\ &= cn^2 \lg(n) - cn^2 \lg(2) + n^2 \\ &= cn^2 \lg(n) + (1 - c)n^2. \end{aligned}$$

If we now pick as $c < 1$, then

$$T(n) = \Omega\left(cn^2 \lg(n)\right).$$

Substitution Method - Example

Use the substitution method to show that

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

is $O\big(n \lg(n)\big)$.

We are trying to prove that $T(n) \leq cn \lg(n)$. So we start by assuming that the bound holds for $\frac{n}{2}$:

$$\begin{aligned} T(n) &\leq c\left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) \\ &\leq cn \lg\left(\frac{n}{2}\right) + n \\ &= cn \lg(n) - cn \lg(2) + n \\ &\leq cn \lg(n). \end{aligned}$$

Since the last step holds for any $c > 1$, we are done. □

Greedy Method – Solving Problems

Greedy Algorithm

→ A greedy algorithm is one which finds optimal solution at each and every stage with the hope of finding the global optimum at the end.

→ Greedy algorithms do not always give optimal solutions but for many problems they do

Example Algorithms:

1. Algorithm for Generation of Huffman Code
2. " Activity selection problem
3. " Knapsack Problem
4. Dijkstra's single Source Shortest problem

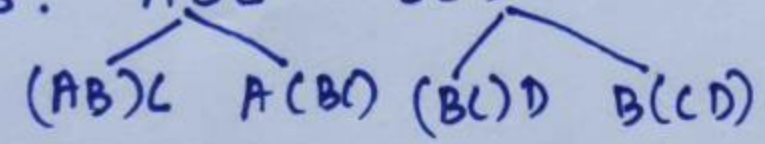
Greedy Algorithms:

A	B	C	D
5×3	3×4	4×2	2×6

$\ell = 2$: AB BC CD

$\ell = 2$: AB (60) BC (24) CD (48)

$\ell = 3$: ABC BCD

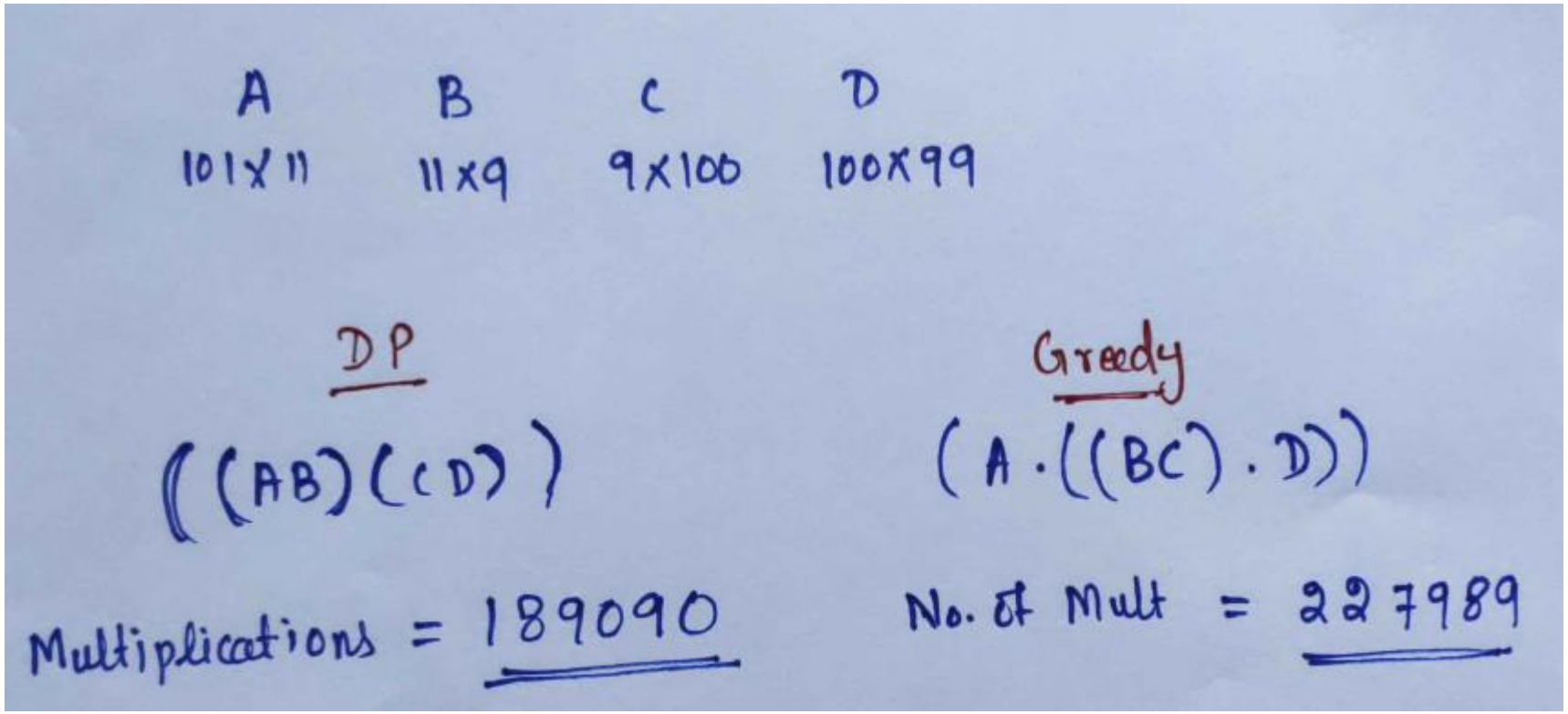


$\ell = 3$: A(BC) (54) (BC)D (60)

$\ell = 4$: ABCD

$\ell = 3$: (A.(BC)).D

114


A
101x11
B
11x9
C
9x100
D
100x99
DP
((AB)(CD))
Multiplications = 189090
Greedy
(A.((BC).D))
No. of Mult = 227989

Greedy Method – Fractional Knapsack Problem

Greedy Method:

Knapsack Problem:

Objects:	0:	1	2	3	4	5	6	7
Profits:	p:	10	5	15	7	6	18	3
Weights:	w:	2	3	5	7	1	4	1

n : 7

C : 15 Kg.

Bag (knapsack)

Objective : Need to fill the bag with available objects without exceeding its capacity and with maximum profit.

Problem : Maximization of Profit
↳ type of optimization.

Constraint : 1. Objects are divisible.
2. Should not exceed bag capacity.

Problem is known as Fractional Knapsack Problem.

→ For Optimization problems, greedy approach is best suitable.

→ Optimization is a Computational problem in which the object is to find the best solution from all possible solutions.

→ Problem to find a solution in the feasible region which has the minimum (or maximum) value of the objective function.

Method - 1:

(Selecting the objects with maximum profit)

Objects	Profit	weight	Remaining weight

Total Profit =

Objects:	0:	1	2	3	4	5	6	7
Profits:	P:	10	5	15	7	6	18	3
Weights:	w:	2	3	5	7	1	4	1

n : 7

C : 15 Kg.

Bag (knapsack)

Method - 1:

(Selecting the objects with maximum profit)

Objects	Profit	weight	Remaining weight
6	18	4	15 - 4 = 11
3	15	5	11 - 5 = 6
1	10	2	6 - 2 = (4)
4	(7)	(7) 4	4 - 4 = 0
	$7 \times \frac{4}{7} = 4$		

Total Profit = 47

Method - 2:

(Selecting the objects with minimum weight)

Objects	Profit	weight	Remaining weight

Total Profit =

Objects :	0 :	1	2	3	4	5	6	7
Profits :	P :	10	5	15	7	6	18	3
Weights :	w :	2	3	5	7	1	4	1

n : 7

C : 15 Kg.

Bag (knapsack)

Method - 2:

(Selecting the objects with minimum weight)

Objects	Profit	weight	Remaining weight
5	6	1	15 - 1 = 14
7	3	1	14 - 1 = 13
1	10	2	13 - 2 = 11
2	5	3	11 - 3 = 8
6	18	4	8 - 4 = (4)
3	(15)	(5) 4	4 - 4 = 0
	$15 \times \frac{4}{5} = 12$		

Total Profit = 54

Method - 3 :

(Selecting Object with maximum profit Per kg)

O :	1	2	3	4	5	6	7
P :	10	5	15	7	6	18	3
w :	2	3	5	7	1	4	1

P/w :

Objects	Profit	weight	Remaining weight

Total Profit =

Method-3:

(Selecting Object with maximum profit Per kg)

O:	1	2	3	4	5	6	7
P:	10	5	15	7	6	18	3
w:	2	3	5	7	1	4	1
P/w:	5	1.67	3	1	6	4.5	3

Objects	Profit	weight	Remaining weight
5	6	1	15 - 1 = 14
1	10	2	14 - 2 = 12
6	18	4	12 - 4 = 8
3	15	5	8 - 5 = 3
7	3	1	3 - 1 = (2)
2	(5)	(3) 2	2 - 2 = 0
	$5 \times \frac{2}{3} = 3.33$		

Total Profit = 55.33

Method - 3:

(Selecting Object with maximum profit Per kg)

O:	1	2	3	4	5	6	7
P:	10	5	15	7	6	18	3
w:	2	3	5	7	1	4	1
P/w:	5	1.67	3	1	6	4.5	3

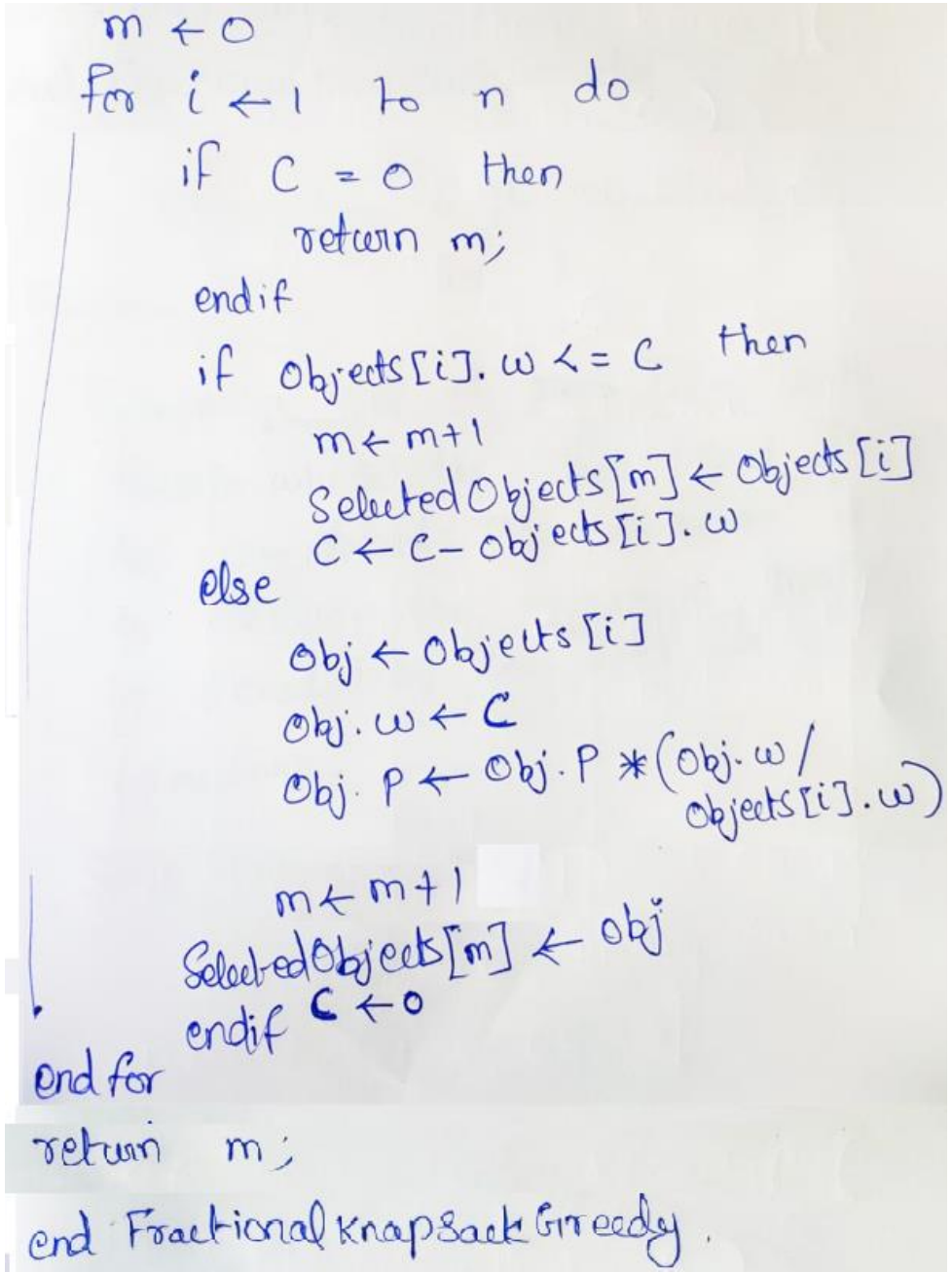
Algorithm FractionalKnapsackGreedy(
Objects[1..n], n, c)

Input: Objects[1..n] with profit & weight.
Size, n
Bag capacity, c

Output: Selected Objects[1..m] and m, number of selected objects.

```
//Calculating Profit per kg
for i ← 1 to n do
    Objects[i].pw ← Objects[i].P / Objects[i].w
endfor.
```

Sort the Objects[1..n] based on pw in descending order.



```
m ← 0
for i ← 1 to n do
    if C = 0 then
        return m;
    endif
    if objects[i].w <= C then
        m ← m+1
        SelectedObjects[m] ← objects[i]
        C ← C - objects[i].w
    else
        obj ← objects[i]
        obj.w ← C
        obj.P ← obj.P * (obj.w /
                         objects[i].w)
        m ← m+1
        SelectedObjects[m] ← obj
        C ← 0
    endif
end for
return m;
end FractionalKnapSackGreedy.
```

Greedy Method –
Job Sequencing with Deadlines Problem

Job Sequencing with Deadlines

Problem:

Given a set of ~~prob~~ Jobs with their Profit and deadline ~~to~~ within which the Jobs need to be completed. The objective is to obtain the maximum profit by scheduling the Jobs in timeline.

Example:

Jobs:	j :	1	2	3	4	5
Profits:	p :	20	15	10	5	1
Deadlines:	d :	2	2	1	3	3

n : 5

Solution:

Maximum Time a Job can Finish: (3)

So no. of time slots : (3)

0 — 1 — 2 — 3 ⇒ Time Slots.

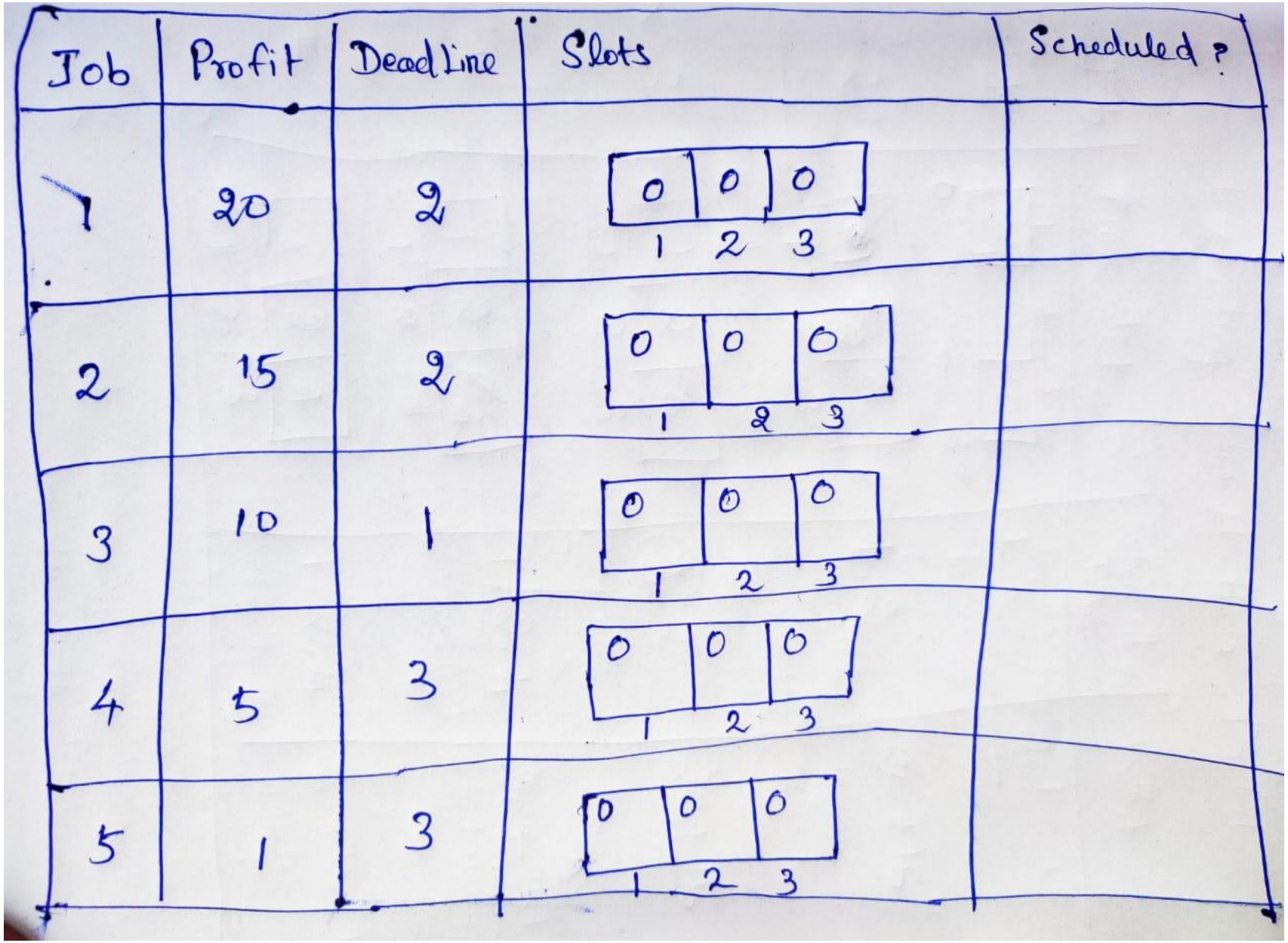
Each Job take 1 unit amount of time.

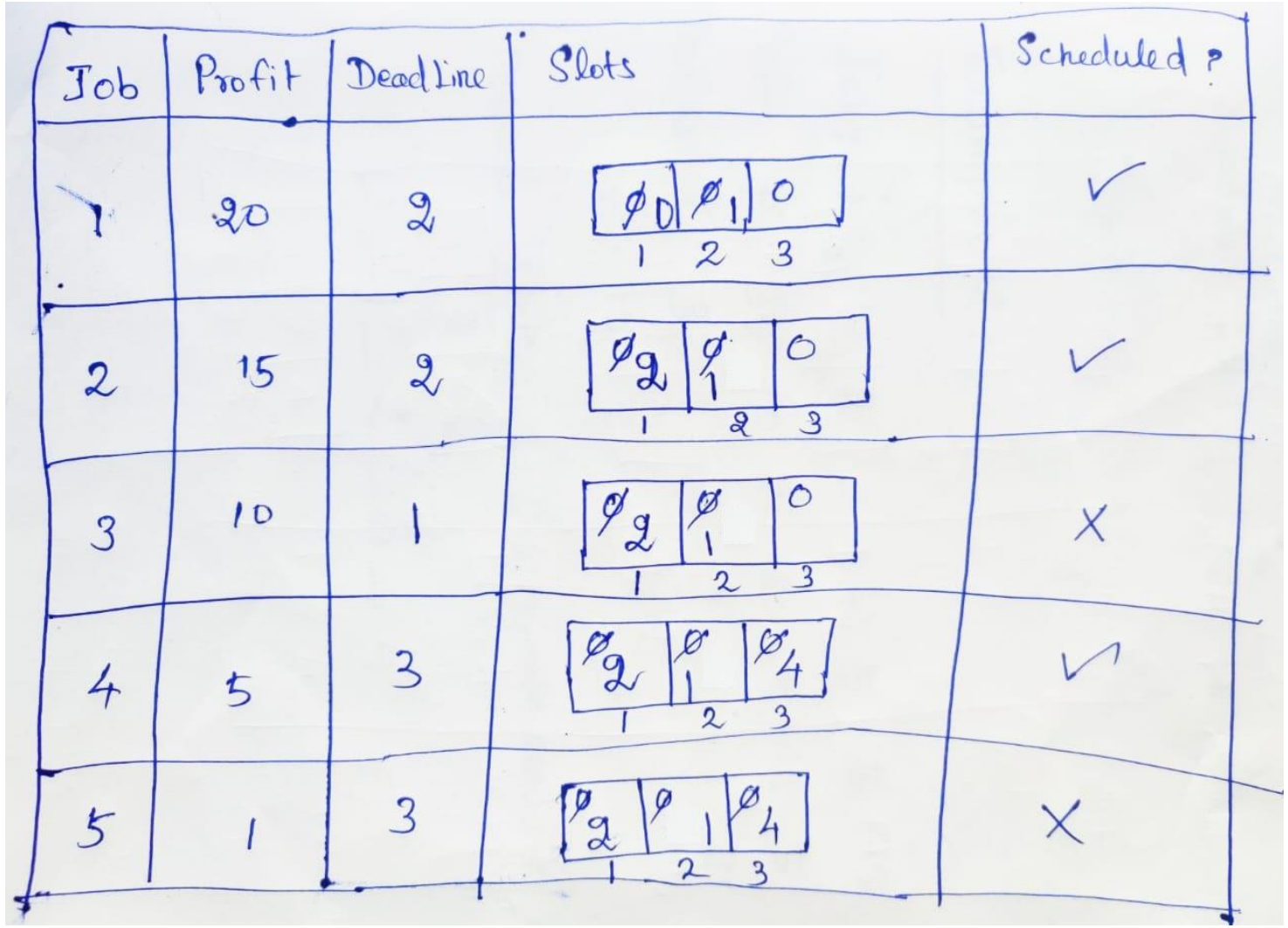
Approach:

→ Select the job with maximum profit.

→ Schedule it, if slot is available with in the Job's deadline.

So, need to Sort the jobs in descending order of their Profit.


Job
Profit
Deadline
Slots
Scheduled ?
1
20
2
2
15
2
3
10
1
4
5
3
5
1
3
0 0 0
1 2 3



Scheduled Jobs : $\{J_2, J_1, J_4\}$

Total Profit : 40

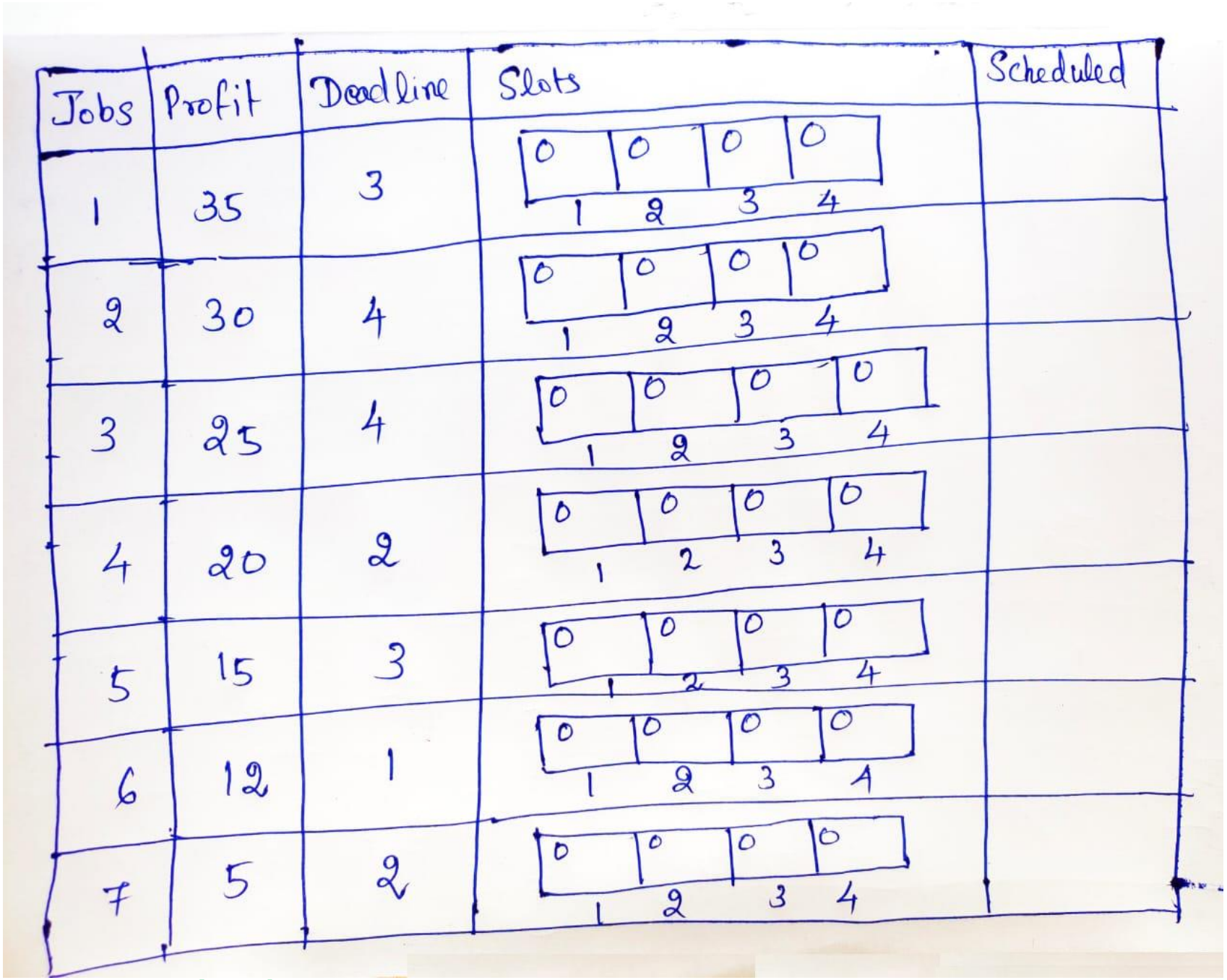
Example 2:

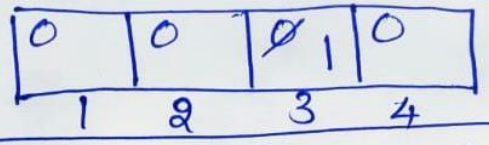
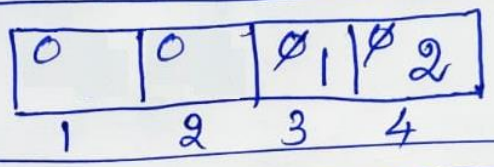
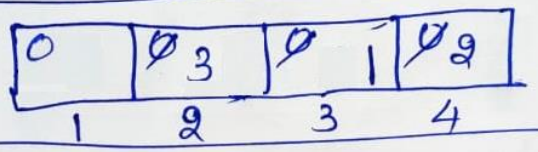
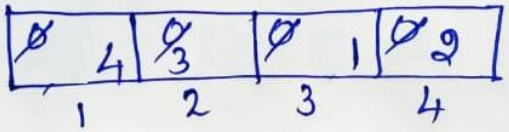
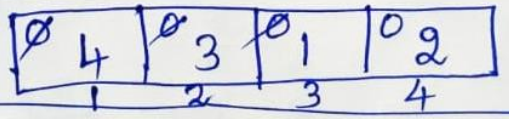
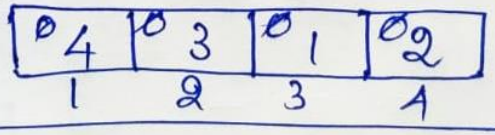
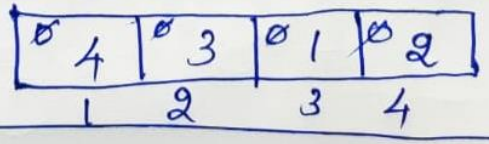
Jobs: j :	1	2	3	4	5	6	7
Profits: p :	35	30	25	20	15	12	5
Deadlines: d :	3	4	4	2	3	1	2

n : 7

Maximum Slots : 4 (maximum deadline)

0 _ 1 _ 2 _ 3 _ 4


Jobs
Profit
Deadline
Slots
Scheduled
1
35
3
2
30
4
3
25
4
4
20
2
5
15
3
6
12
1
7
5
2
0
1
2
3
4

Jobs	Profit	Deadline	Slots	Scheduled
1	35	3		✓
2	30	4		✓
3	25	4		✓
4	20	2		✓
5	15	3		✗
6	12	1		✗
7	5	2		✗

Scheduled Jobs: $\{J_4, J_3, J_1, J_2\}$

Total Profit : 110

```
Algorithm JobSequencingGreedy(
                    Jobs[1..n], n)

Input : Jobs[1..n] with profit and
                    deadlines.
        No. of Jobs, n
Output : Slot[1..m] and the size m.

// Finding maximum slot.
m ← maximum(Jobs[1..n].deadline)

Sort the Jobs[1..n] in descending
    order of its profit.
```

```
Let     Slot[1..m] to maintain
    the order of Jobs to be done.
For i ← 1  to m do
    Slot[i] ← 0
end for
```

```
for i ← 1 to n do
    for j ← Jobs[i].deadline to 1
                    downwards do
        if Slot[j] = 0 then
            Slot[j] ← i
            break from loop.
        end if
    end for
end for.
return
end JobSchedulingGreedy.
```

Summary

- ✓ Algorithm Design Model
- ✓ Efficiency of Algorithms – Space & Time
- ✓ Asymptotic Notations – Big-Oh, Big-Omega and Theta
- ✓ Problem Solving Strategies – Brute-Force, Divide & Conquer, Greedy – Heuristic Algorithms
- ✓ Solving Recurrences – Recursion Tree Method
- ✓ Solving Recurrences – Iterative Method
- ✓ Solving Recurrences – Using Master Theorem
- ✓ Brute-Force – Travelling Salesperson Problem
- ✓ Divide & Conquer – Maximum Sub Array
- ✓ Greedy – Fractional Knapsack Problem
- ✓ Greedy – Job Sequencing