Recurrence Relation

Recursion

- The process in which a function calls itself directly or indirectly is called **recursion**
- Note:- (i) Every recursive algorithm must have a base case to simplify to.

  (ii) When a function is called, some information needs to be saved in order to return the calling module back to its original state (the state it was in before the call).
- Example:- Factorial of number

```
factorial(n)
{
   i= 1;
   fact= 1;
   while(i<=n)
   {
     fact= fact * i;
     i = i+1;
   }
   return (fact);
}
```

```
factorial (int n)
{

 if (n==0)
    return 1;
 else
    return n* factorial (n - 1);

}
```

- How recursion works? https://www.youtube.com/watch?v=ygK0YON10sQ

Introduction to Recurrence Relation

- A recurrence relation is used to compute complexity of recursive algorithms/functions.
- A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs.
- There are four methods for solving Recurrence:

  (1) Substitution Method

  (2) Iteration Method

  (3) Recursion Tree Method

  (4) Master Theorem

Substitution Method

The Substitution Method Consists of two main steps:

Step 1:- Guess the Solution.

Step 2:- Use mathematical induction to find the constants and show that the solution works.

Example 1.1 Solve equation T(n)= T(n/2)+1. We have to show that it is bounded by *O(log n)*

Recurrence relation $T(n) = T(n/2) + 1$ ------- *(i)*

Assume solution will be $T(n) = O(\log_2 n)$

We have to show that for some constant c

$T(n) \leq c \log_2 n$ ---------- *(ii)*

Put *(ii)* in given Recurrence relation *(i)*

$T(n) \leq c \log_2(n/2) + 1$

$\leq c \log_2(n/2) + 1$

$= c \log_2 n - c \log_2 2 + 1$

$= c \log_2 n - 1 + 1$

$\leq c \log_2 n$ for $c \geq 1$

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

Example 1.2 Solve equation T(n)= 2T(n/2)+ n

Recurrence Equation $T(n) = 2T(n/2)+ n , \quad n>1$ ---------- *(i)*

We guess the solution is O (n ($\log_2 n$)).

Thus for constant c, $T(n) \leq c\, n \log_2 n$ ---------- *(ii)*

Put *(ii)* in given Recurrence Equation *(i)*

Now,

$T(n) \leq 2\, c(n/2) \log_2 (n/2) + n$

$\leq c\, n\log_2 n - c\, n \log_2 2 + n$

$= c\, n \log_2 n - n (c \log_2 2 - 1)$, assume c=1

$\leq c\, n \log_2 n$ for ($c \geq 1$)

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

Iteration Method

- It means to expand the recurrence and express it as a summation of terms of n and initial condition
- We iteratively “unfold” the recurrence until we “see the pattern”
- The iteration method does not require making a good guess like the substitution method (but it is often more involved than using induction).

Example 2.1 Solve recurrence relation with iteration method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n-1) & \text{if } n>1 \end{cases}$$

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

$= 2[2T(n-2)]$

$T(n) = 2^2T(n-2)$

$= 2^2[2T(n-3)]$

$= 2^3T(n-3)$

$T(n-1) = 2\,T((n-1)-1)$
$= 2\,T(n-2)$

$T(n-2) = 2\,T((n-1)-2)$
$= 2\,T(n-3)$

Repeat the procedure for i times

$T(n) = 2^i\,T(n-i)$ -------- *(i)*

Assume $T(n-i) = T(1)$,

So, $n-i = 1$ or $i = n-1$ ------ *(ii)*

Put *(ii)* in *(i) and we get* →

$T(n) = 2^{n-1}\,T(1)$

$= 2^{n-1} \cdot 1$ **w.k.t T (1) =1**

$= 2^{n-1}$

Example 2.2 Solve recurrence relation with iteration method

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

$T(n) = 2\,T(n/2) + n$

$= 2[2T(n/2^2) + n/2\,] + n$

$T(n) = 2^2\,T(n/2^2) + n + n$

$= 2^2\,[2\,T((n/2^3) + n/2^2)\,] + 2n$

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

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

$T(n/2) = 2\,T((n/2^2) + n/2)$

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

Repeat the procedure for i times

$T(n) = 2^i\,T(n/2^i) + i\,n$ -------- *(i)*

Assume $T(n/2^i) = T(1)$,

So, $n/2^i = 1$

$n = 2^i$ or $i = \log n$ ------ *(ii)*

Put *(ii)* in *(i) and we get* →

$T(n) = 2^i\,T(n/2^i) + i\,n$

$= n\,T(1) + n \log n$

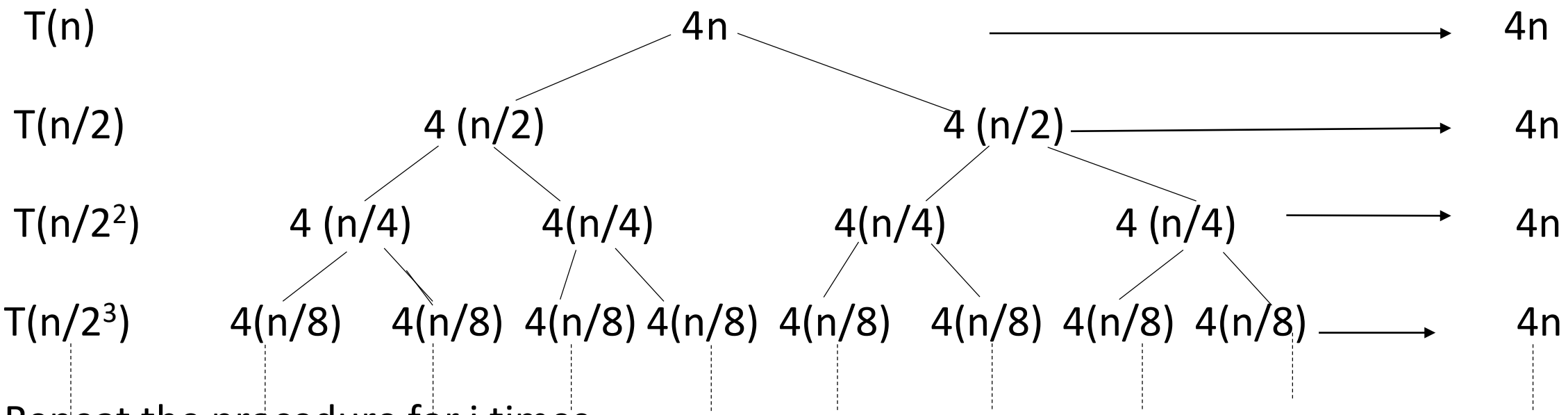
$= n \cdot 1 + n \log n$ w.k.t $T(1) = 1$

$= O(n \log n)$

Recursion Tree Method

- A pictorial representation of an iteration method which is in form of a tree where at each level nodes are expanded.
- Root value is second term which is not recursive in nature.
- In Recursion tree, each root and child represents the cost of a single sub problem.
- We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

Example 3.1 T (n) = 2T(n/2) + 4n



Repeat the procedure for i times

$T(n/2^i)$

Assume $T(n/2^i) = T(1)\ n^2$

So, $n/2^i = 1$

$2^i = n$ or $i = \log n$

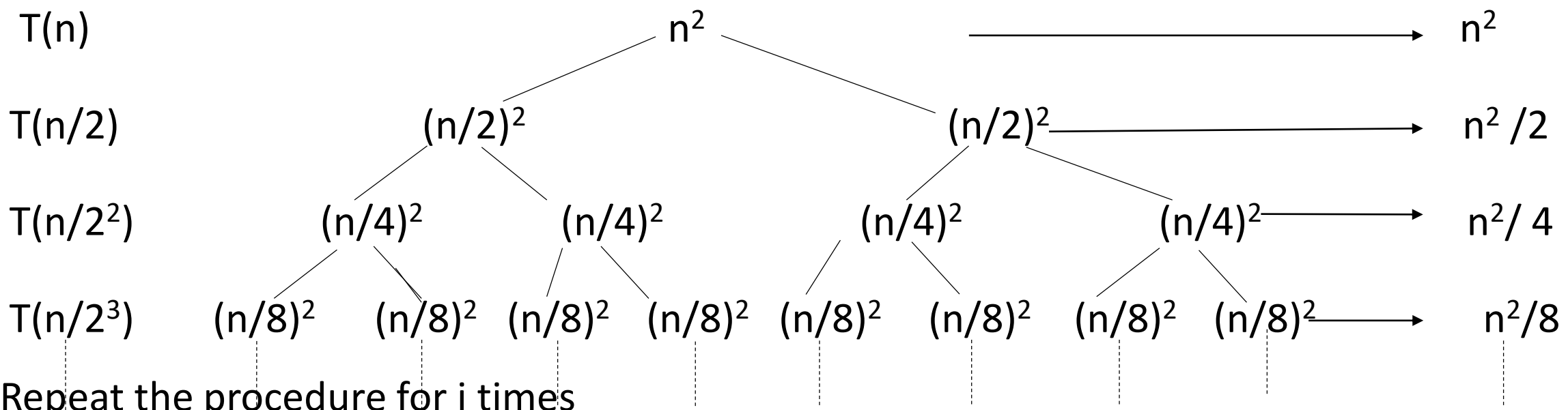
$= 4n+4n+4n+\ldots\ldots+4n+\ldots.+$ upto i times

$$= \sum_{i=0}^{\log n} 4n$$

$$= 4n \sum_{i=0}^{\log n} 1$$

$= 4n (\log n + 1)$ => **O(n log n)**

Example 3.2 $T(n) = 2T(n/2) + n^2$



Repeat the procedure for i times

$T(n/2^i)$

Assume $T(n/2^i) = T(1)\ n^2$

So, $n/2^i = 1$

$2^i = n$ or $i = \log n$

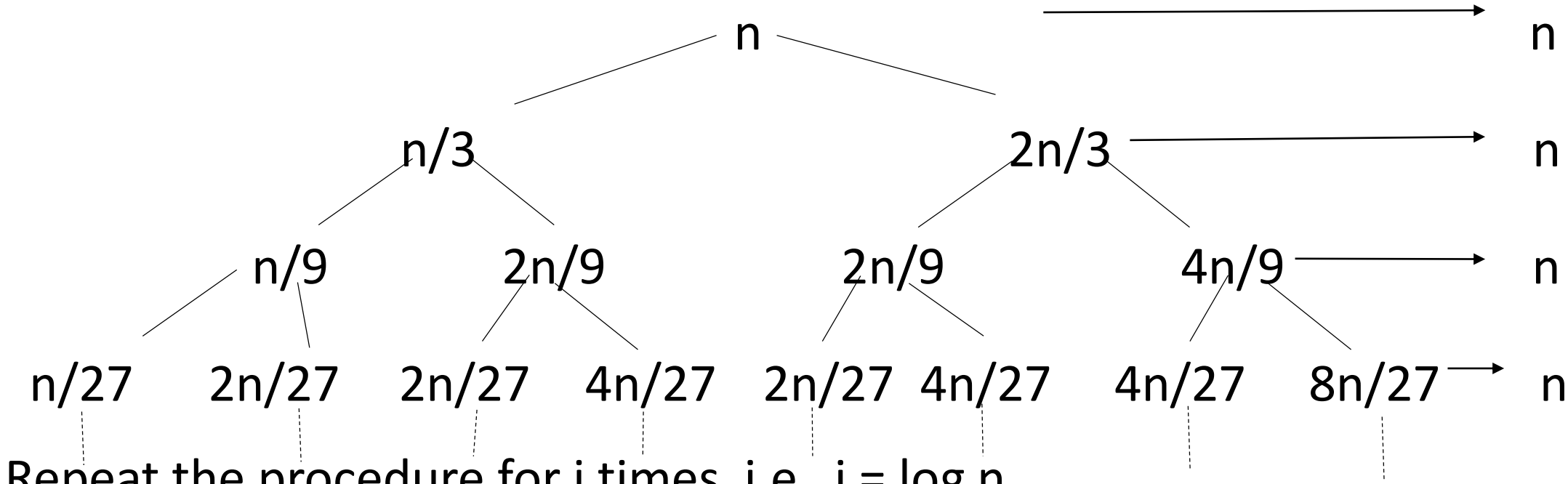
$= n^2 + n^2/2 + n^2/4 + n^2/8 + \ldots\ldots\ldots +$ log n times

$= n^2\ (1+1/2+1/4+1/8+\ldots\ldots)$

$= n^2 \sum_{i=0}^{\infty} (1/2^i)$ w.k.t $\sum_{i=0}^{\infty} a\,r^i = a/1-r\ ,\ |r|<1$

$= n^2 \left(\frac{1}{1 - 1/2} \right)$ $= \mathbf{2\ n^2} = \mathbf{O(n^2)}$

Example 3.3 T (n) = T(n/3) + T(2n/3) + n



Repeat the procedure for i times i.e i = log n

= n+n+n+n…….. + log n
log n
= n ∑ 1 = n log n = O(n log n)
i= 0

Master Theorem

- It works only for following type of recurrences or for recurrences that can be transformed to following type:-

$$\mathbf{T(n) = a\,T(n/b) + f(n)}$$

where **$a>=1$, $b>1$ and $f(n)= \text{theta}(n^c \log^p n)$**

(i) n is the size of the problem

(ii) a is the number of sub problems in the recursion

(iii) n/b is the size of each sub problem. (Here it is assumed that all sub problems are essentially the same size)

(iv) f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the sub problems.

There are following 3 cases :-

Case I : If $f(n) = O(n^c)$ where $c < \log_b a$

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

Case II : If $f(n) = O(n^c)$ where $c = \log_b a$

(i) If $P > -1$ then **$T(n) = \text{theta}(n^c \log^{p+1} n)$**

(ii) If $P = -1$ then **$T(n) = \text{theta}(n^c \log \log n)$**

(iii) If $P < -1$ then **$T(n) = \text{theta}(n^c)$**

Case III : If $f(n) = O(n^c)$ where $c > \log_b a$

(i) If $P >= 0$ then **$T(n) = \text{theta}(n^c \log^p n)$**

(ii) If $P < 0$ then **$T(n) = O(n^c)$**

Example 1: $T(n) = 8T(n/2) + 1000\ n^2$

Solution : $a = 8,\ b = 2,\ f(n) = 1000\ n^2$

$f(n) = O(n^c)$ where $c=2$

$\log_b a = \log_2 8 = 3$

$c < \log_b a$ i.e. $2<3$ (Satisfy Case I)

So, $T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 8})$ **$= O(n^3)$**

Example 2.1: T(n)= 2T(n/2) + 10n

Solution : a = 2, b = 2, f(n)= 10n

where c = 1 and P=0

$\log_b a = \log_2 2 = 1$

$c = \log_b a$ i.e. 1=1 (Satisfy Case II.i i.e. P>-1)

So, $T(n) = O(n^c \log^{p+1} n)$

$= O(n^1 \log^1 n)$ **$= O(n \log n)$**

Example 2.2: T(n)= 2T(n/2) + n log n

Solution : a = 2, b = 2, f(n)= n log n

where c = 1 and P=1

$\log_b a = \log_2 2 = 1$

$c = \log_b a$ i.e. 1=1 (Satisfy Case II)

Here P=1, so it satisfy Case (II.i i.e. P>-1)

Time complexity is **$T(n) = \text{theta } (n^c \log^{p+1} n)$**

$= \text{theta } (n^1 \log^{1+1} n)$

$= \text{theta } (n \log^2 n)$

Example 2.3: $T(n)= 4T(n/2) + (n^2/ \log n)$

Solution : $a = 4,\ b = 2,\ f(n)= (n^2/ \log n) = n^2 \log^{-1} n$

where $c = 2$ and $P=-1$

$\log_b a = \log_2 4 = 2$

$c = \log_b a$ i.e. $2=2$ (Satisfy Case II)

Here $P=-1$, so it satisfy Case (II.ii i.e. $P=-1$)

Time complexity is **$T(n) = \text{theta } (n^c \log \log n)$**

$= \text{theta } (n^2 \log \log n)$

Example 2.4: $T(n)= 8T(n/2) + (n^3/ \log^2 n)$

Solution : $a = 8$, $b = 2$, $f(n)= (n^3/ \log^2 n) = n^2 \log^{-2} n$

where $c = 3$ and $P=-2$

$\log_b a = \log_2 8 = 3$

$c = \log_b a$ i.e. $3=3$ (Satisfy Case II)

Here $P=-2$, so it satisfy Case (II.iii i.e. $P<-1$)

Time complexity is **$T(n) = \text{theta } (n^c)$**

$= \text{theta } (n^3)$

Example 3.1: $T(n)= 2T(n/2) + n^2$

Solution : a = 2, b = 2, $f(n)= n^2$

where c = 2 and P=0

$\log_b a = \log_2 2 = 1$

$c > \log_b a$ i.e. $2 > 1$ (Satisfy Case III.i i.e. P>=0)

So, T(n) = **theta ($n^c \log^p n$)**

= theta ($n^2 \log^0 n$) (w.k.t. $\log^0 n = 1$)

= theta (n^2)

Example 3.2: $T(n)= 4T(n/2) + (n^3 / \log^2 n)$

Solution : $a = 4$, $b = 2$, $f(n)= (n^3 / \log n)$

where $c = 3$ and $P=-2$

$\log_b a = \log_2 2 = 1$

$c > \log_b a$ i.e. $3 > 1$ (Satisfy Case III.ii i.e. $P<0$)

So, $T(n) = \mathbf{O(n^c)}$

$\mathbf{= O(n^3)}$

Inadmissible Equations in master theorems

The following equations cannot be solved using the master's theorem:-

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

a is not a constant, the number of sub problem should be fixed.

(ii) $T(n) = 0.5T(n/2) + n$

a <1, cannot have less than 1 sub problems

(iii) $T(n) = 64\ T(n/8) – n^2 \log n$

In f(n), combination time should be positive but in this it is negative