

Designing Techniques of Algorithms, Recurrence Relations, Design & Analysis of Sorting Algorithms, Linear Sorting, Searching and Red Black Tree

Design and Analysis of Algorithms



Why Designing Techniques



- ✓ A problem can be solved in various different approaches
- ✓ Some approaches deliver much more efficient results than others.
- ✓ The algorithm Designing technique is used to measure the effectiveness and performance of the algorithms.

List of several popular design approaches

What we'll briefly understand today

01

Divide and Conquer Strategy

02

Greedy Technique

03

Dynamic Programming

04

Branch and Bound

05

Backtracking Strategy



Divide and Conquer Strategy

What you should know

01

Dividing the problem into sub-problems

02

Solve every subproblem individually, recursively

03

Combining them for the final answer

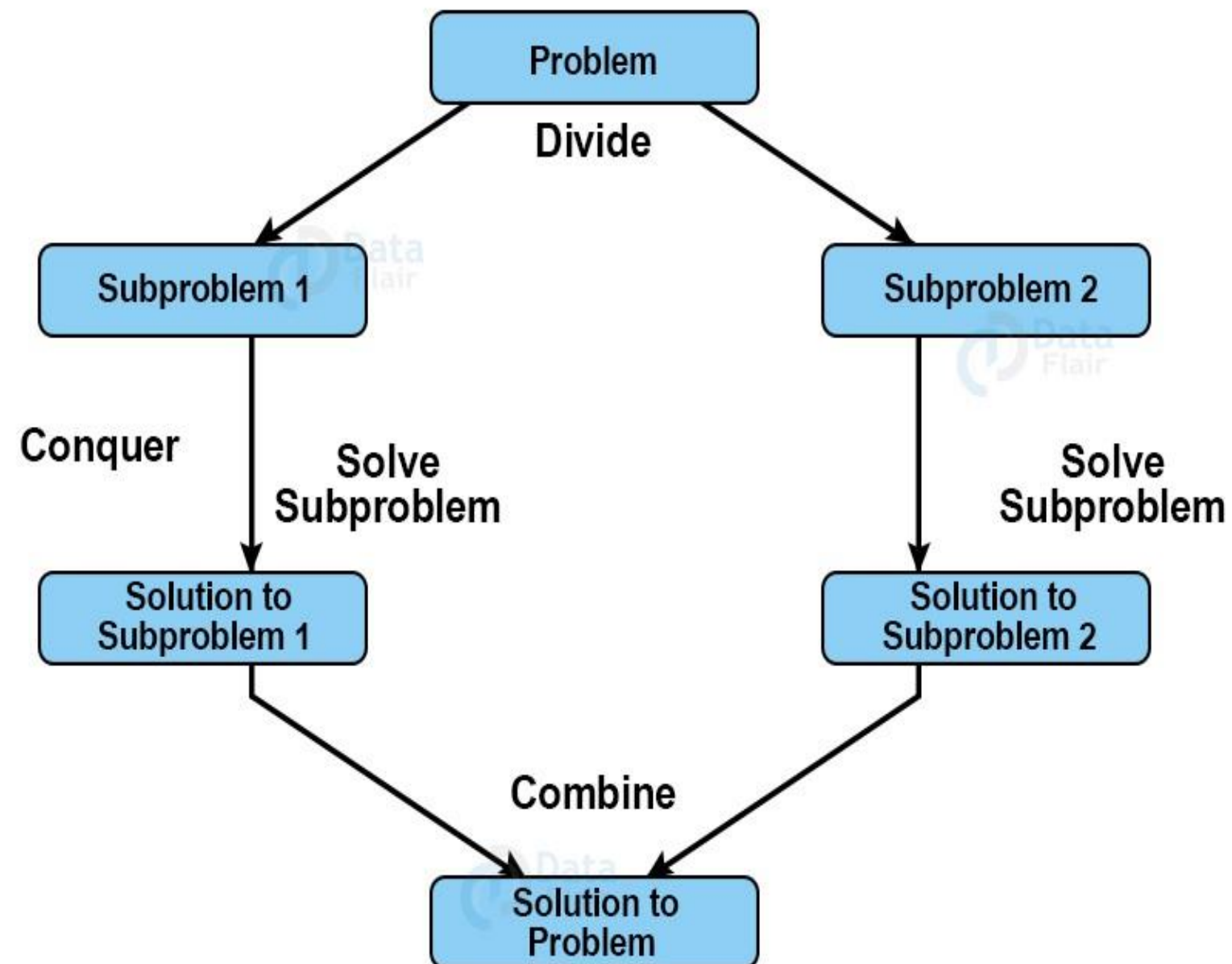
04

Follows top-down approach



Divide and Conquer Strategy

What you should know



This technique can be divided into the following three parts:

Divide: This involves dividing the problem into smaller sub-problems.

Conquer: Solve sub-problems by calling recursively until solved.

Combine: Combine the sub-problems to get the final solution of the whole problem.

The strategy of designing algorithms has two fundamentals

01

Recurrence formula

02

Stopping Conditions

Divide and Conquer Strategy

What you should know

01

Recurrence Formula/Equation

Function has been defined in terms of same function with some changes in the argument.

02

Stopping Conditions

When you divide the problem by divide and conquer strategy, you need to understand for how long you have to keep dividing. You have to put a stopping condition to stop your recursion steps.



Examples

For Example, the Worst Case Running Time $T(n)$ of the MERGE SORT Procedures is described by the recurrence

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 2T(n/2) + O(n) & \text{if } n>1 \end{cases}$$



Applications of Divide and Conquer

Binary search

01

Merge sort

02

Quicksort

03

Strassen's
algorithm

04

Advantage of Divide and Conquer Strategy



- ✓ Divide and conquer successfully solved one of the biggest problems of the mathematical puzzle world, the tower of Hanoi.
- ✓ You might have a very basic idea of how the problem is going to be solved but dividing the problem makes it easy since the problem and resources are divided.
- ✓ It is very much faster than other algorithms.
- ✓ The divide and conquer algorithm works on parallelism. Parallel processing in operating systems handles them very efficiently.
- ✓ The divide and conquer strategy used cache memory without occupying much main memory. Executing problems in cache memory which is faster than main memory.

Disadvantage of Divide and Conquer Strategy

- ✓ Most of the divide and conquer design uses the concept of recursion therefore it requires high memory management.
- ✓ It may crash the system if recursion is not performed properly.



Recursion & Recurrence Relations



What does recursion means

- ✓ A function calling itself is called **recursion** & corresponding function is called as **recursive function**.

```
fact(int n)
{
    if (n <= 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```



How recursion works



Example- Sum of first n natural numbers

Approach(1) - Simply adding one by one

Algorithm Sum (A, n)

```
{  
    S=0;  
    for ( i=0; i<n; i++)  
    {  
        S= S + A[i];  
    }  
    return S;  
}
```

Approach(2) – Recursive adding

$\text{Sum}(n) = n + \text{sum}(n-1)$

$\text{Sum}(5) = 5+4+3+2+1$

$\text{Sum}(5) = 5 + \text{Sum}(4)$

$\text{Sum}(4) = 4 + \text{Sum}(3)$

$\text{Sum}(3) = 3 + \text{Sum}(2)$

$\text{Sum}(2) = 2 + \text{Sum}(1)$

$\text{Sum}(1) = 1$

Algorithm Sum (n)

```
{  
    if(n==1)  
        return 1;  
    else  
        return n + sum(n-1);  
}
```

What does recursion relation means

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

← Base/Stopping Condition

- ✓ A recurrence relation is an equation or inequality that describes a function in terms of its values on smaller inputs.
- ✓ Used to reduce complicated problems to an iterative process based on simpler versions of the problem
- ✓ $T(n)$ term is used to define the time complexity in recurrence relation analysis.



There are four methods for solving Recurrence



- ✓ 1. Back Substitution/ Iteration Method
- ✓ 2. Recursion Tree Method
- ✓ 3. Master Method
- ✓ 4. Substitution Method

How to analyze the Recurrence Algorithm

Recursion Tree Method

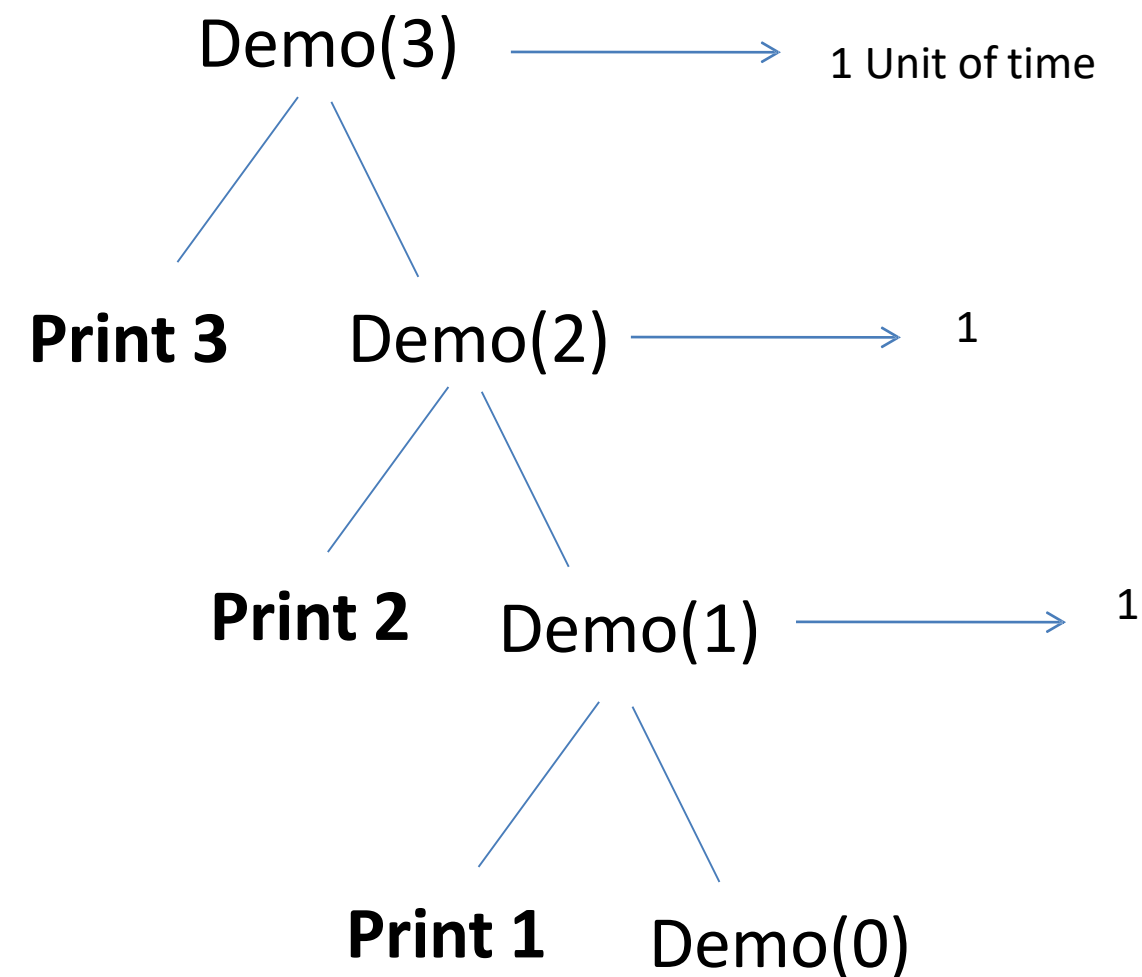
Draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels

Example- Analyze the time complexity of this algorithm

Algorithm Demo(n)

```
{  
  If (n>0)  
  {  
    Print " n";  
    Demo(n-1);  
  }  
}
```

Lets put n=3



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

Number of times Print statements is executed = 3
Total number of function call = 3+1



How to write and solve a Recurrence Equation

Example- Analyze the time complexity of this algorithm

Algorithm Demo(n) _____ $T(n)$

{

 If ($n > 0$)

 {

 Print "n"; _____ 1

 Demo($n-1$); _____ $T(n-1)$

 }

}

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

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + 1 & \text{if } n>0 \end{cases}$$



How to write and solve a Recurrence Equation (Back Substitution/ Iteration Method)

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

Calculate the $T(n-1)$ and then substitute the value

Put $n = n-1$ in Equation (1)

Substitute back

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

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

Put $n = n-2$ in Equation (1)

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

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

Continued for k times

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

Recall this equation

$$S(n) = n + s(n-1)$$

If I know $s(n-1)$,
then I can easily calculate $s(n)$

We know that $T(0) = 1$
(base Condition)

Assume that $n-k=0$
Then $k=n$

$$T(n) = T(0) + k * 1$$

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

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

Iteration Method

Expand the recurrence by solving the smaller terms and express it as a summation of terms of n and initial condition.



How to write and solve a Recurrence Equation

Master Method (cookbook method)



Master Method is a direct way to get the solution.



The master method works for following type of recurrences or for recurrences that can be transformed to following type. (*)




$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b \geq 1$

a = number of sub-problems in the recursion

n/b = size of each sub-problem


$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + 1 & \text{if } n>1 \end{cases}$$

(*) we can also derive the master theorem for decreasing function.



How to analyze the Recurrence Algorithm

Substitution Method

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + 1 & \text{if } n>0 \end{cases}$$

We guess the solution as $T(n) = O(n)$

Now we use mathematical induction to prove our guess. We need to prove that $T(n) \leq cn$ for all value of $n \geq n_0$ and $c > 0$

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= c(n-1) + 1 = cn - c + 1 = cn + 1 - c \end{aligned}$$

$$T(n) = cn + 1 - c \leq cn \text{ for all } c \geq 1$$

Lets Put $c=1$

$$1*n + 1 - 1 \leq 1*n \quad n=n$$

Put $c=2$

$$2*n + 1 - 2 \leq 2*n$$

$$2n - 1 < 2n$$



How to analyze the Recurrence Algorithm

```
Algorithm Demo(n)
{
  If (n>0)
  {
    for (i=0; i<n; i++)
    {
      print " n";
    }
    Demo(n-1);
  }
}
```



How to analyze the Recurrence Algorithm

```
Algorithm Demo(n)  $\longrightarrow$  T(n)
{
  If (n>0)
  {
    for (i=0; i<n; i++)
    {
      print " n";  $\longrightarrow$  n
    }
    Demo(n-1);  $\longrightarrow$  T(n-1)
  }
}
```

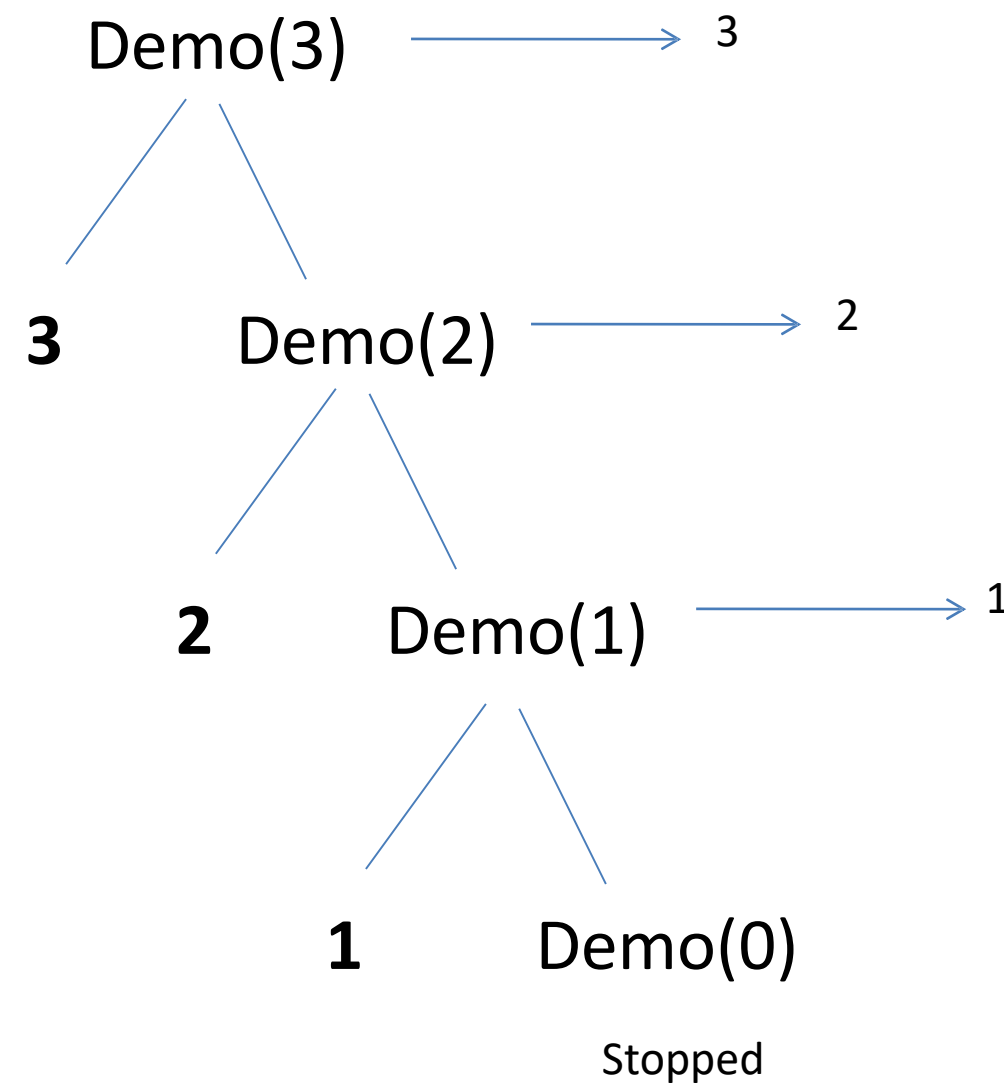
Recurrence Relation $T(n) = T(n-1) + n$

For n elements, number of times Print statements is executed $= 1+2+3+4+5+\dots+n = \frac{n(n+1)}{2} = n^2$

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

Lets put $n=3$

Recurrence Tree
Method



Number of times Print statements is executed $= 1+2+3$

Total number of function call $= 3+1$



How to analyze the Recurrence Algorithm

Iterative Method

Recurrence Relation $T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + n & \text{if } n>0 \end{cases}$ (Base Equation)

Put $n = n-1$

$$\begin{aligned} T(n-1) &= T((n-1)-1) + n-1 \\ &= T(n-2) + n-1 \end{aligned}$$

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

Put $n = n-2$ in the base equation

$$\begin{aligned} T(n-2) &= T((n-2)-1) + n-2 \\ &= T(n-3) + n-2 \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-3) + n-2 + n-1 + n \\ &= T(n-4) + n-3 + n-2 + n-1 + n \end{aligned}$$

Continued for k times

$$T(n) = T(n-k) + n-k+1 + n-k+2 + \dots + n-1 + n$$

$$T(n) = T(n-k) + n-(k-1) + n-(k-2) + \dots + n-1 + n$$

For $n=0$, $T(0)=1$, i.e. $n-k=0 \rightarrow n=k$

$$\begin{aligned} T(n) &= T(0) + n-(n+1) + n-(n-2) + \dots + n-1 + n \\ &= T(0) + 1 + 2 + 3 + \dots + n-1 + n \\ &= T(0) + n(n+1)/2 \\ &= 1 + (n^2 + n)/2 \\ &= O(n^2) \end{aligned}$$

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



How to analyze the Recurrence Algorithm

Algorithm Demo(n)

```
{  
  If (n>0)  
  {  
    print " n";  
    Demo(n-1);  
    Demo(n-1);  
  }  
}
```

How to analyze the Recurrence Algorithm

Algorithm Demo(n) $\longrightarrow T(n)$

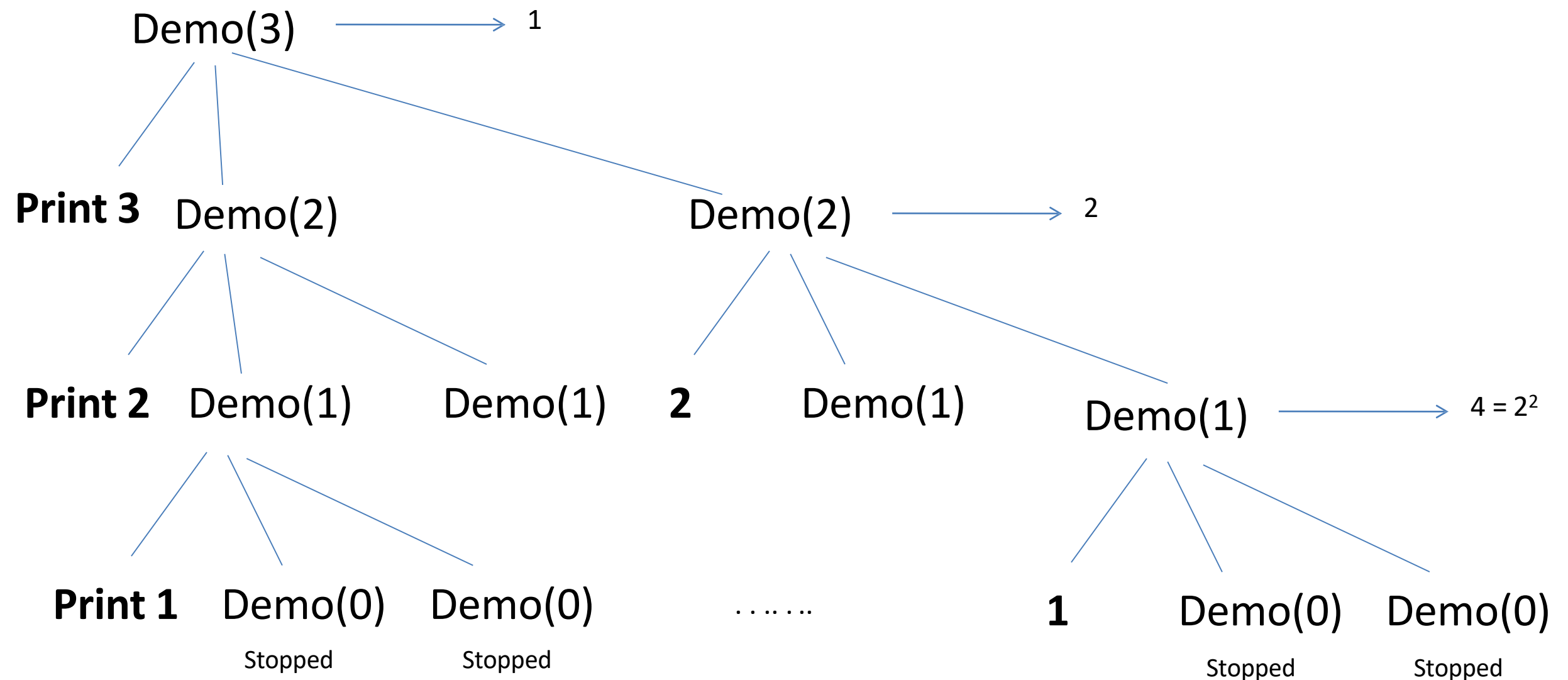
```

{
  If (n>0)
  {
    print " n";  $\longrightarrow 1$ 
    Demo(n-1);  $\longrightarrow T(n-1)$ 
    Demo(n-1);  $\longrightarrow T(n-1)$ 
  }
}

```

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

Lets put $n=3$ **Recurrence Tree Method**



Number of times Print statements is executed $= 1 + 2 + 2^2$

For n elements, number of times Print statements is executed $= 1 + 2 + 2^2 + 2^3 + 2^4 + \dots + 2^n = 2^{n+1} - 1 = 2^n$

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

Using GP Series formula: $a + ar^2 + ar^3 + ar^4 + \dots + ar^n = \frac{ar^{n+1} - 1}{r - 1}$

How to analyze the Recurrence Algorithm

Recurrence Relation

$$T(n) = 2T(n-1) + 1 \text{ for } n > 0$$

$$= 1 \quad \text{for } n = 0$$

Iteration Method

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

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

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

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

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

.

.

.

$$= 2^k T(n-k) + 2^{k-1} + \dots + 2^2 + 2 + 1$$

For $n=0$, $T(0)=1$, it means that $n-k=0 \rightarrow n=k$

$$2^n + 2^{n-1} + \dots + 2^2 + 2 + 1 = 2^{n+1} - 1 = O(2^n)$$



Quick Revision of Time Complexity

(Decreasing Functions)

$$T(n) = T(n-1) + 1 \text{ for } n > 0$$
$$= 1 \text{ for } n = 0$$

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

$$T(n) = T(n-1) + n \text{ for } n > 0$$
$$= 1 \text{ for } n = 0$$

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

$$T(n) = 2T(n-1) + 1 \text{ for } n > 0$$
$$= 1 \text{ for } n = 0$$

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

$$T(n) = 2T(n-1) + n \text{ for } n > 0$$
$$= 1 \text{ for } n = 0$$

$$T(n) = O(n \cdot 2^n)$$



How to analyze the Recurrence Algorithm

(Dividing Functions)

Algorithm Demo(n)

{

If ($n > 1$)

{

Print " n ";

Demo($n/2$);

}

}



How to analyze the Recurrence Algorithm

(Dividing Functions)

Algorithm Demo(n) \longrightarrow T(n)

{

If (n>1)

{

Print "n"; \longrightarrow 1

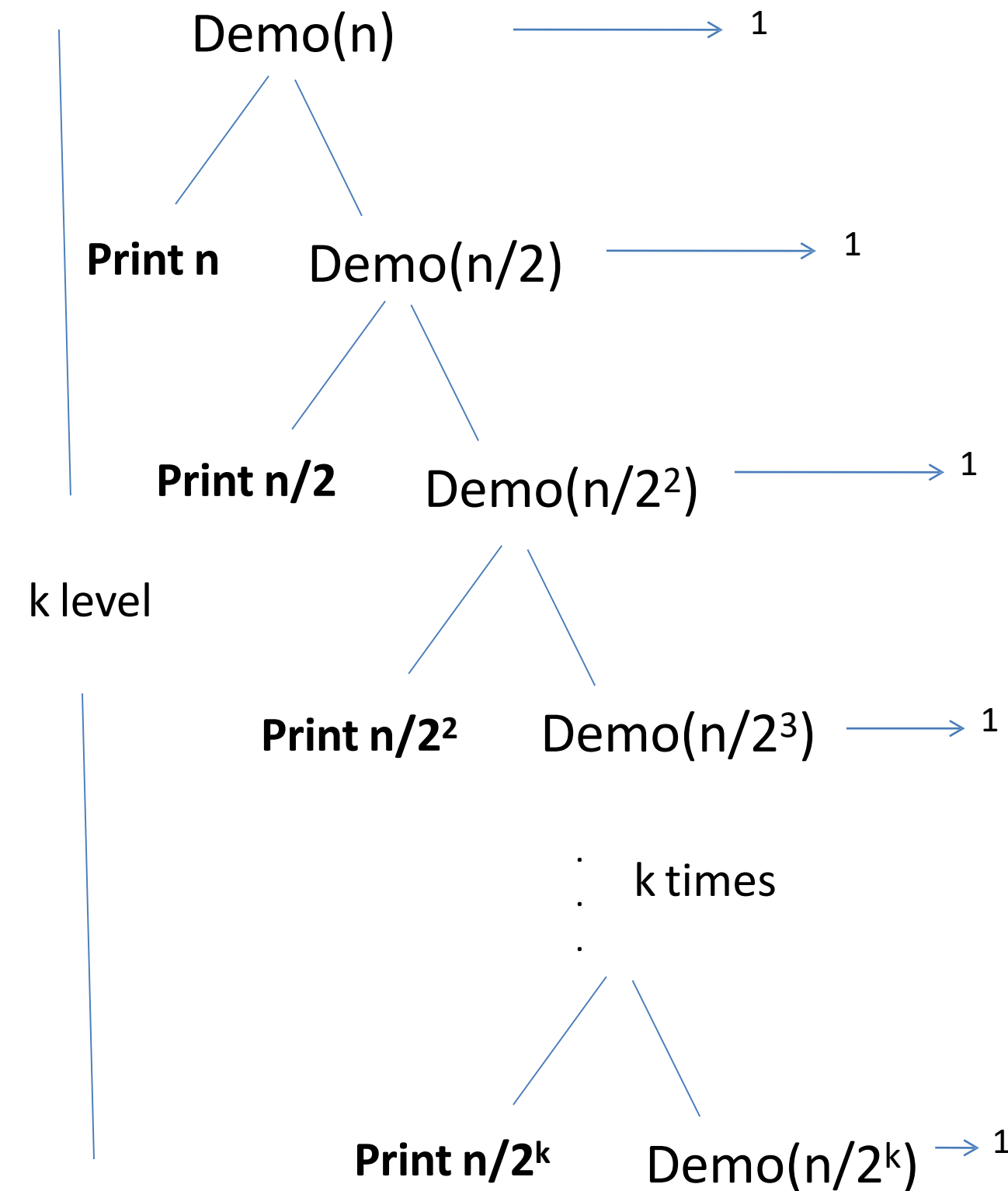
Demo(n/2); \longrightarrow T(n/2)

}

}

Stopping Condition: $n/2^k=1$
 $= n=2^k$
 $= k= \log n = T(n)= O(\log n)$

Recurrence Tree Method



How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

$$T(n) = T(n/2) + 1 \text{ for } n > 1$$
$$= 1 \quad \text{for } n = 1$$

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

Iterative Method

$$\begin{aligned} T(n) &= T(n/2) + 1 && \text{Put } n=n/2 \\ &= T(n/2^2) + 1 + 1 && T(n/2) = T(n/2^2) + 1 \\ &= T(n/2^3) + 1 + 1 + 1 \\ &\cdot \\ &\cdot \\ &\cdot \\ &= T(n/2^k) + K * 1 \\ T(n) &= T(1) + k && \text{put } n/2^k = 1 \\ &= 1 + k && = n = 2^k \\ &= \log n + 1 && = k = \log n \\ &= O(\log n) \end{aligned}$$



How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

$$T(n) = 2T(n/2) + n \text{ for } n > 1$$

$$= 1$$

$$\text{for } n = 1$$

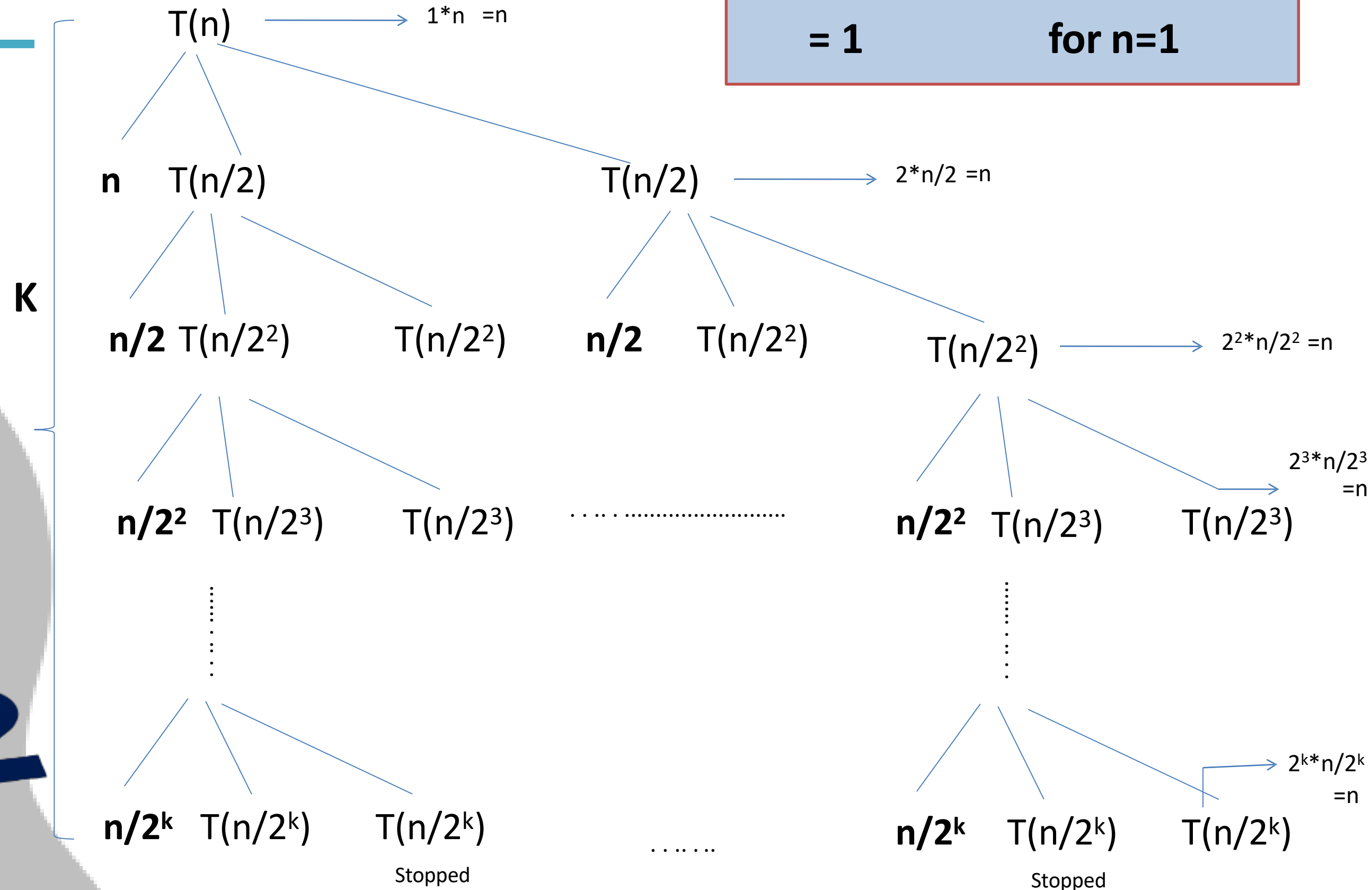


How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

$$T(n) = 2T(n/2) + n \text{ for } n > 1$$
$$= 1 \quad \text{for } n = 1$$



How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

$$T(n) = 2T(n/2) + n \text{ for } n > 1$$

$$= 1 \quad \text{for } n = 1$$

$$T(n) = n + n + n + n + n + \dots + n \quad (k \text{ times})$$
$$= n * k$$

or

$$T(n) = \text{number of level or height of tree} * \text{time taken at each level}$$
$$= k * (n)$$
$$= n * k$$

$$T(n) = n * \log n$$

As we know that stopping condition

$$n/2^k = 1$$

$$n = 2^k \quad (\text{apply log})$$

$$\Rightarrow k = \log n$$



How to analyze the Recurrence Algorithm

(Dividing Functions)

Recurrence Relation

$$T(n) = T(n/2) + n \text{ for } n > 1$$

$$= 1 \quad \text{for } n = 1$$



How to analyze the Recurrence Algorithm

(Dividing Functions)

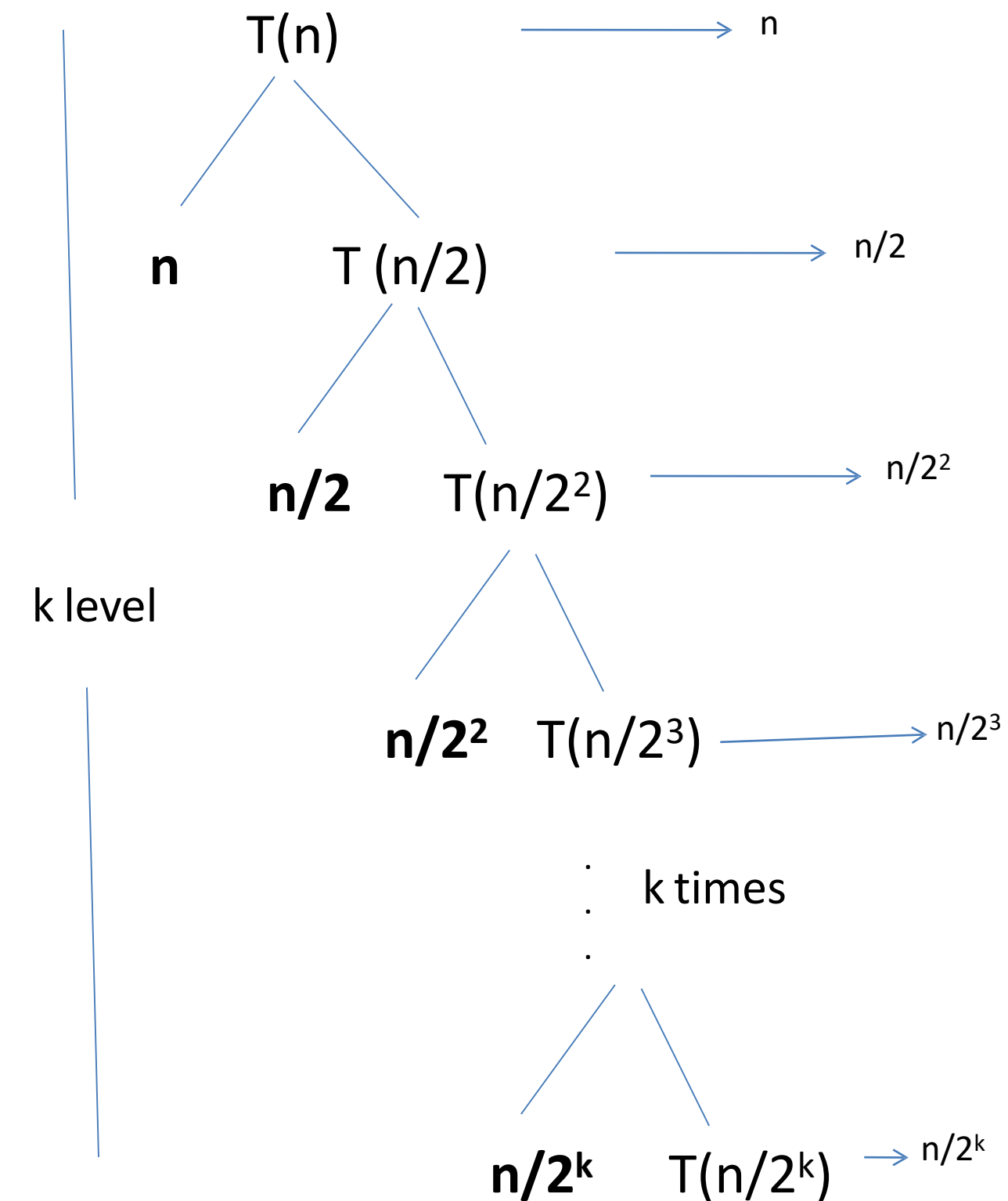
Recurrence Relation

$$T(n) = T(n/2) + n \text{ for } n > 1$$

$$= 1 \quad \text{for } n = 1$$

$$\begin{aligned} T(n) &= n + n/2 + n/2^2 + n/2^3 + \dots + n/2^k \\ &= n (1 + 1/2 + 1/2^2 + 1/2^3 + \dots + 1/2^k) \\ &= n * 1 \\ &= O(n) \end{aligned}$$

Recurrence Tree Method



How to analyze the Recurrence Algorithm

(Dividing Functions)

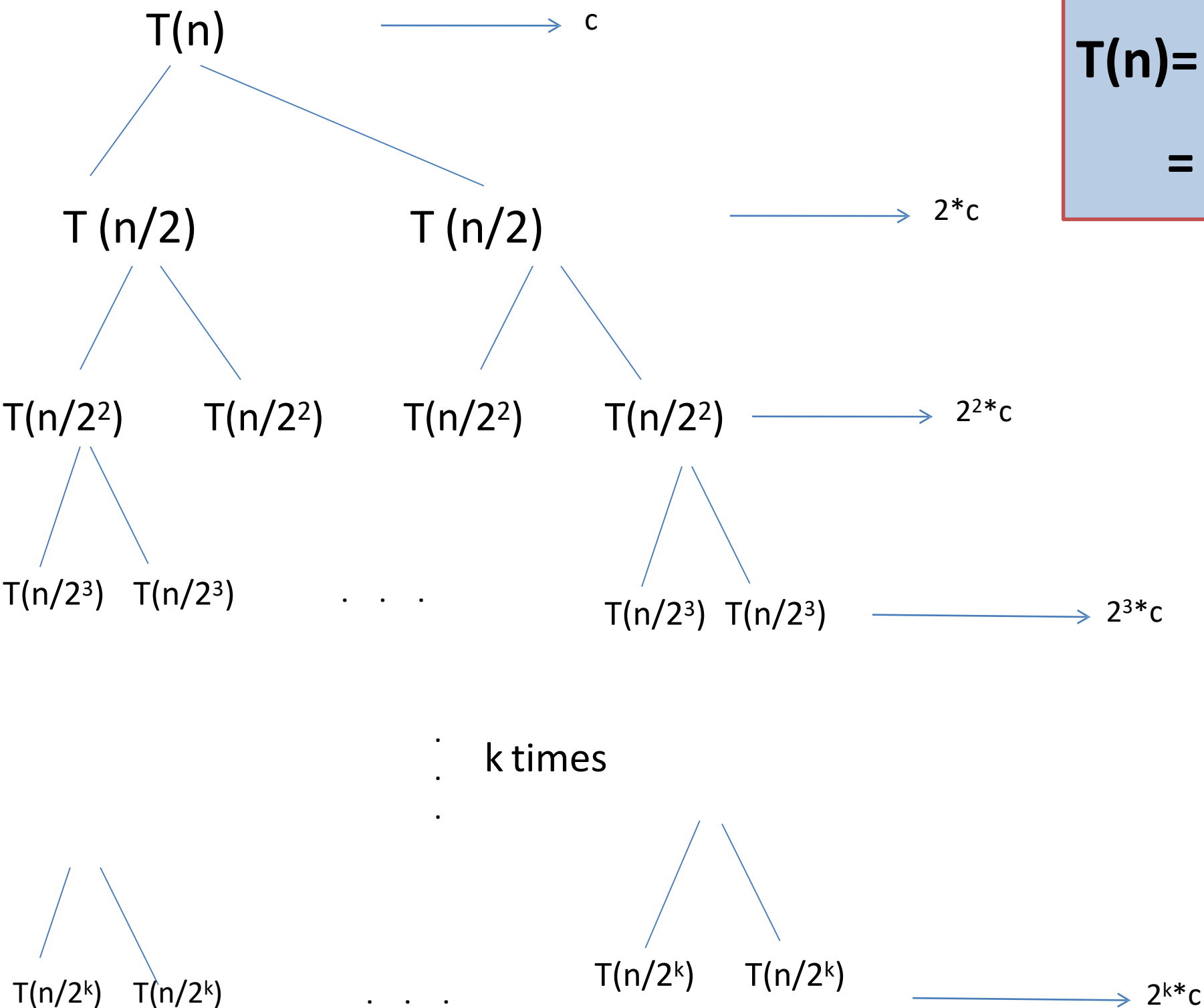
Recurrence Relation

$$T(n) = 2T(n/2) + c \text{ for } n > 1$$
$$= 1 \quad \text{for } n = 1$$



How to analyze the Recurrence Algorithm

(Dividing Functions)



Recurrence Relation

$$T(n) = 2T(n/2) + c \text{ for } n > 1$$

$$= 1 \quad \text{for } n = 1$$

$$\begin{aligned}
 T(n) &= c + 2*c + 2^2*c + 2^3*c + \dots + 2^k*c \\
 &= c (1 + 2 + 2^2 + 2^3 + \dots + 2^k) \\
 &= c (2^{k+1} - 1) \\
 &= c * 2^k \\
 &\text{put } k = \log n \\
 &= c * 2^{\log n} \\
 &= c * n \quad (2^{\log n} = n^{\log 2})
 \end{aligned}$$

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

As we know that stopping condition

$$n/2^k = 1$$

$$n = 2^k \quad (\text{apply log})$$

$$\Rightarrow k = \log n$$


Quick Revision of Time Complexity

(Dividing Functions)

$$T(n) = T(n/2) + 1 \text{ for } n > 1$$
$$= 1 \text{ for } n = 1$$

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

$$T(n) = 2T(n/2) + n \text{ for } n > 1$$
$$= 1 \text{ for } n = 1$$

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

$$T(n) = T(n/2) + n \text{ for } n > 1$$
$$= 1 \text{ for } n = 1$$

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

$$T(n) = 2T(n/2) + c \text{ for } n > 1$$
$$= 1 \text{ for } n = 1$$

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



How to analyze the Recurrence Algorithm

(Dividing Functions)



Master Method (cookbook method)



Master Method is a direct way to get the solution.



The master method works for following type of recurrences or for recurrences that can be transformed to following type. (*)

$$T(n) = aT(n/b) + f(n)$$

n = size of input ,

a = number of sub-problems in the recursion , n/b = size of each sub-problem.

$f(n)$ = cost of the work done outside the recursive call (extra work-done), which includes the cost of dividing the problem and cost of merging the solutions

Here, $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function.

How to analyze the Recurrence Algorithm

(Dividing Functions)

Master Method (cookbook method)

$$T(n) = aT(n/b) + f(n)$$

Calculate $n^{\log_b a}$

Case -1 If $f(n) > n^{\log_b a}$ Then $T(n) = O(f(n))$

Case -2 If $f(n) < n^{\log_b a}$ Then $T(n) = O(n^{\log_b a})$

Case -3 If $f(n) = n^{\log_b a}$ Then $T(n) = O(n^{\log_b a} \log n)$ or $O(f(n) * \log n)$



How to analyze the Recurrence Algorithm

(Using Master Theorem)

Solve it using Master Theorem

$$T(n) = aT(n/b) + f(n)$$

Example -1

$$\begin{aligned} T(n) &= 2T(n/2) + n \text{ for } n > 1 \\ &= 1 \quad \quad \quad \text{for } n = 1 \end{aligned}$$



How to analyze the Recurrence Algorithm

(Using Master Theorem)

Solve it using Master Theorem

$$T(n) = aT(n/b) + f(n)$$

$$a=2, b=2, f(n) = n$$

Calculate $n^{\log_b a}$

$$n^{\log_2 2} = n$$

$f(n) = n$ is given

Case-3 condition satisfied

Example -1

$$\begin{aligned} T(n) &= 2T(n/2) + n \text{ for } n > 1 \\ &= 1 \quad \text{for } n = 1 \end{aligned}$$

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

Case -3 If $f(n) = n^{\log_b a}$
Then $T(n) = O(f(n) * \log n)$

$$\begin{aligned} \text{Hence, } T(n) &= O(f(n) * \log n) \\ &= O(n * \log n) \end{aligned}$$



Examples Exercise of Master Theorem

Example -2

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

Master Theorem

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



Examples Exercise of Master Theorem

Master Theorem

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

Example -2

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

$$a=4, b=2, f(n) = n^3$$

$$\text{Calculate } n^{\log_b a} = n^{\log_2 4} = n^2$$

$$f(n) = n^3 \text{ is given}$$

Case-1 condition is satisfied

$$\begin{aligned} \text{Hence, } T(n) &= O(f(n)) \\ &= O(n^3) \end{aligned}$$

Example -3

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



Examples Exercise of Master Theorem

Example -2

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

$$a=4, b=2, f(n) = n^3$$

$$\text{Calculate } n^{\log_b a} = n^{\log_2 4} = n^2$$

$f(n) = n^3$ is given

Case-1 condition is satisfied

$$\begin{aligned}\text{Hence, } T(n) &= O(f(n)) \\ &= O(n^3)\end{aligned}$$

Master Theorem

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

Example -3

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

$$a=7, b=2, f(n) = n^2$$

$$\text{Calculate } n^{\log_b a} = n^{\log_2 7}$$

$f(n) = n^2$ is given

Case-2 condition is satisfied

$$\begin{aligned}\text{Hence, } T(n) &= O(n^{\log_b a}) \\ &= O(n^{\log_2 7})\end{aligned}$$



Examples Exercise of Master Theorem

Example -4

$$T(n) = T(n/4) + n$$

Master Theorem

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



Examples Exercise of Master Theorem

Example -4

$$T(n) = T(n/4) + n$$

$$a=1, b=4, f(n) = n$$

$$\text{Calculate } n^{\log_4 1} = n^0 = 1$$

$f(n) = n$ is given

Case-1 condition is satisfied

$$\begin{aligned} \text{Hence, } T(n) &= O(f(n)) \\ &= O(n) \end{aligned}$$

Master Theorem

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

Example -5

$$T(n) = 4T(7n/3 + 5) + n^2$$



Examples Exercise of Master Theorem

Example -4

$$T(n) = T(n/4) + n$$

$$a=1, b=4, f(n) = n$$

$$\text{Calculate } n^{\log_4 1} = n^0 = 1$$

$f(n) = n$ is given

Case-1 condition is satisfied

$$\begin{aligned} \text{Hence, } T(n) &= O(f(n)) \\ &= O(n) \end{aligned}$$

Master Theorem

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

Example -5

$$T(n) = 4T(7n/3 + 5) + n^2$$

$$a=4, b=3/7, f(n) = n^2$$

$$\text{Calculate } n^{\log_b a} = n^{\log_4 3/7}$$

$f(n) = n^2$ is given

Case-1 condition is satisfied

$$\begin{aligned} \text{Hence, } T(n) &= O(f(n)) \\ &= O(n^2) \end{aligned}$$



Not all recurrence relations can be solved with the use of the master theorem i.e. if

- $T(n)$ is not monotone, ex: $T(n) = \sin n$
- $f(n)$ is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$



Advance Master Theorem (Extended)

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

where n = size of the problem , $a > 1$ and $b > 1$

a = number of sub-problems in the recursion , n/b = size of each sub-problem,

$k \geq 0$ and p is a real number, then,

Case 1- if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

Case -2 if $a = b^k$, then

(a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

(b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log \log n)$

(c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

Case-3 if $a < b^k$, then

(a) if $p \geq 0$, then $T(n) = \theta(n^k \log^p n)$

(b) if $p < 0$, then $T(n) = \theta(n^k)$



Example of Advance Master Theorem

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

Example -1

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



Example of Advance Master Theorem

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

Example -1

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

$$a = 2, b = 2, k = 1, p = 2$$

$$b^k = 2. \text{ So, } a = b^k \text{ [Case 2.(a)]}$$

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

$$T(n) = \theta(n^{\log_2 2} \log^3 n)$$

$$T(n) = \theta(n \log^3 n)$$



Example of Advance Master Theorem

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

Example -2

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



Example of Advance Master Theorem

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

Example -2

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

$$a = 3, b = 2, k = 2, p = 0$$

$$b^k = 4. \text{ So, } a < b^k \text{ and } p = 0 \text{ [Case 3.(a)]}$$

$$T(n) = \theta(n^k \log^p n)$$

$$T(n) = \theta(n^2)$$

