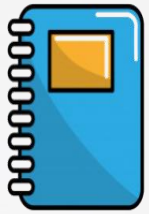
江西理工大学 Jiangxi University of Science and Technology
信息工程学院 School of information engineering


高级算法分析与设计

Advanced Algorithm Analysis and Design

Lecture 06:

Recursion Algorithm

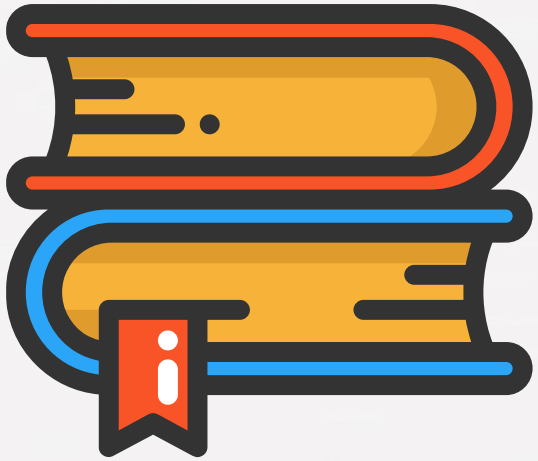
Dr Ata Jahangir Moshayedi

EMAIL: ajm@jxust.edu.cn

Prof Associate ,
School of information engineering Jiangxi university of science and technology, China

LIVE Lecture series

Autumn _2022



高级算法分析与设计

Advanced Algorithm Analysis and Design

LECTURE 06: Recursion Algorithm

Agenda

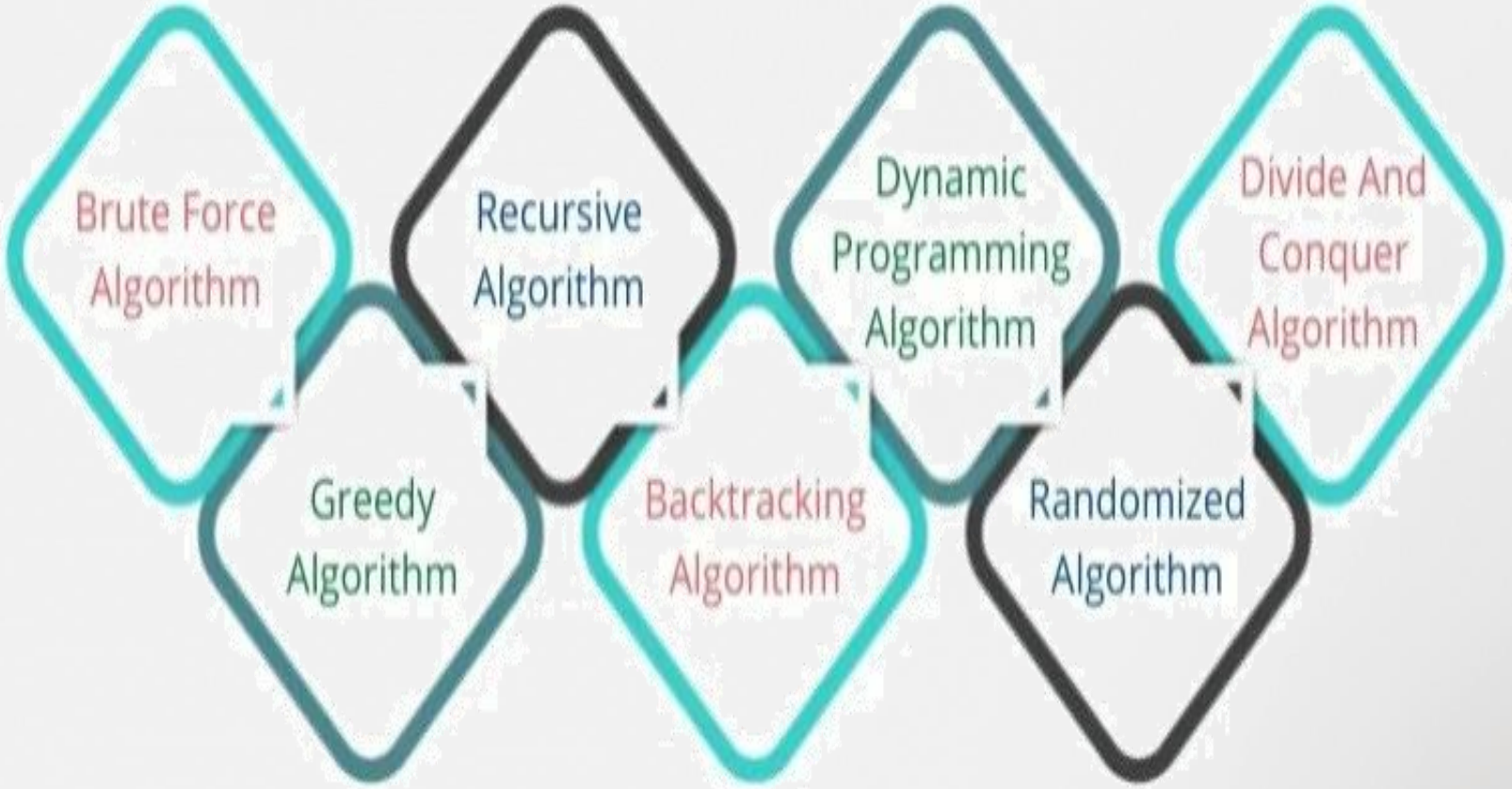
目录

CONTENTS

7 Types Of Algorithms

1. Brute Force Algorithm
2. Recursive Algorithm
3. Dynamic Programming Algorithm
4. Divide and Conquer Algorithm
5. Greedy Algorithm
6. Backtracking Algorithm
7. Randomized Algorithm

1.蛮力算法　　2.递归算法
3.动态规划算法　　4.分而治之算法
5.贪婪算法　　6.回溯算法
7.随机算法



What is Recursion?

- The process in which a function calls itself **directly or indirectly** is called recursion and the corresponding function is called a recursive function.

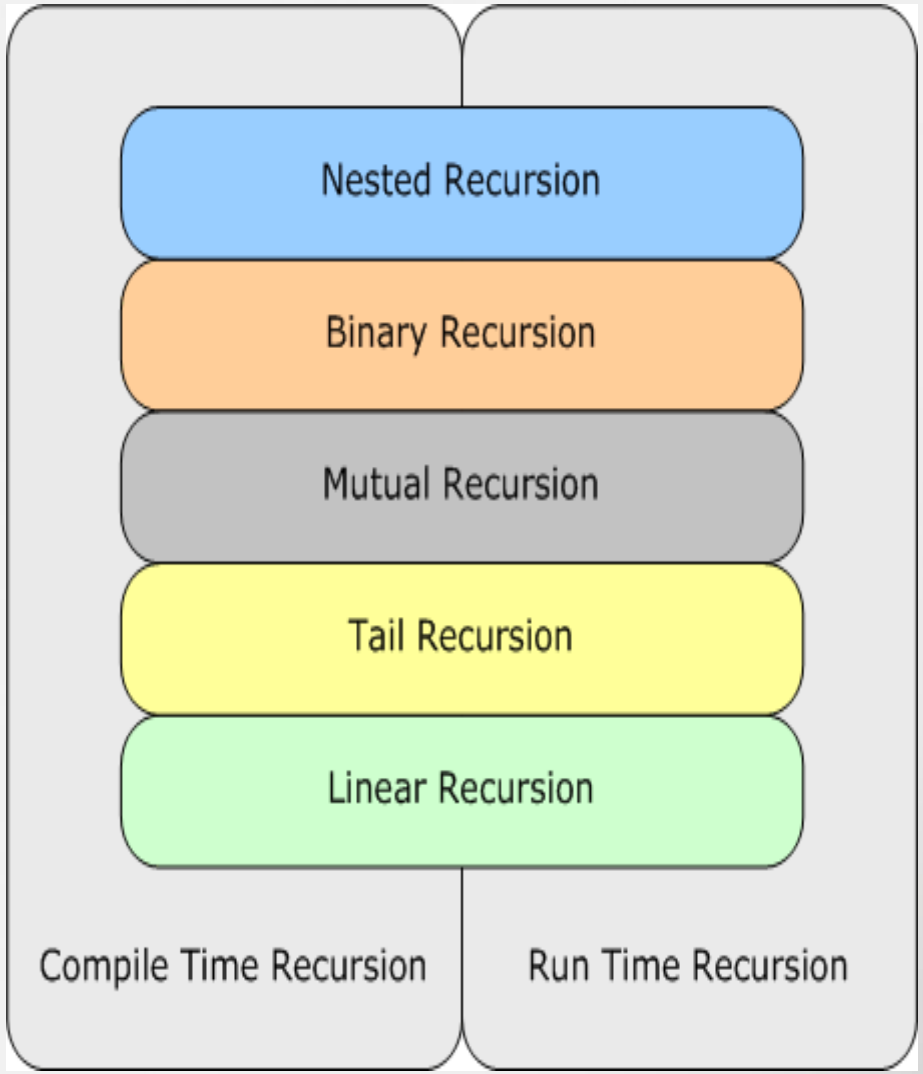
  函数直接或间接调用自身的过程称为递归，对应的函数称为递归函数。

- Using recursive algorithm, certain problems can be solved quite easily.

  采用递归算法，可以很容易地解决某些问题。

- Examples of such problems are **Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.**

  这类问题的例子有:河内塔(TOH)、订单/预购/海报树遍历、图的DFS等。



Need of Recursion

- Recursion is an amazing technique with the help of which **we can reduce the length of our code and make it easier to read and write**.
- It has certain advantages over the iteration technique.
- A task that can be defined with its similar subtask, recursion is one of the best solutions for it.
- For example; The Factorial of a number.

●递归是一种了不起的技术，借助它我们可以减少代码的长度并使其更易于读写。
●与迭代技术相比，它具有某些优势，稍后将讨论。
●递归是可以使用其类似子任务定义的任务，是它的最佳解决方案之一。
●例如;数字的阶乘。

build a recursive algorithm

- To build a recursive algorithm, you will break the given problem statement into two parts. The first one is the base case, and the second one is the recursive step.
  - Base Case: It is nothing more than the simplest instance of a problem, consisting of a condition that terminates the recursive function. This base case evaluates the result when a given condition is met.
  - Recursive Step: It computes the result by making recursive calls to the same function, but with the inputs decreased in size or complexity.

- 要构建递归算法，您需要将给定的问题陈述分为两部分。第一个是基本情况，第二个是递归步骤。
- 基本情况：它只不过是问题的最简单的实例，由终止递归函数的条件组成。此基本情况在 满足给定条件时评估结果。
- 递归步骤：它通过对同一函数进行递归调用来计算结果，但输入的大小或复杂性减小。

Example:
Print sum of n natural numbers using recursion

- Print sum of n natural numbers using recursion.
- This statement clarifies that we need to formulate a function that will calculate the summation of all natural numbers in the range 1 to n. Hence, mathematically you can represent the function as:
- F(n) = 1 + 2 + 3 + 4 + ……..+ (n-2) + (n-1) + n
- It can further be simplified as:

$$F(n) = \sum_{k=1}^{k=n} (k)$$

Breakdown of Problem Statement

$$F(n) = \sum_{k=0}^{k=n} (k)$$

Base Case

```
If(n == 0)
    {
       Return;
    }
```

Recursive step

F(n) = n + F(n-1)

example

- 例如，考虑以下问题陈述：使用递归打印 n 个自然数的总和。这个陈述澄清了我们需要制定一个函数来计算 1 到 n 范围内所有自然数的总和。因此，在数学上，您可以将函数表示为：
- F(n) = 1 + 2 + 3 + 4 + ……..+ (n-2) + (n-1) + n
- 它可以进一步简化为：

$$F(n) = \sum_{k=1}^{k=n} (k)$$

Breakdown of Problem Statement

$$F(n) = \sum_{k=0}^{k=n} (k)$$

Base Case

```
If(n == 0)
    {
        Return;
    }
```

Recursive step

F(n) = n + F(n-1)

Recursive Functions

In programming terms, a recursive function can be defined as a routine that calls itself directly or indirectly.Using the recursive algorithm, certain problems can be solved quite easily.

在编程术语中，递归函数可以定义为直接或间接调用自己的例程。使用递归算法，某些问题可以很容易地解决

- **Towers of Hanoi (TOH)** is one such programming exercise. Try to write an *iterative* algorithm for TOH. Moreover, every recursive program can be written using iterative methods. 《河内塔》(TOH)就是这样一个编程练习。试着为TOH写一个迭代算法。而且，每个递归程序都可以用迭代方法编写。在数学上，递归有助于轻松解决一些难题。
- Mathematically, recursion helps to solve a few puzzles easily.
- **There are three types of recursion : Head Recursion, Tail Recursion, Body Recursion**

递归有三种类型:头递归，尾递归，体递归

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

- 使用不同的输入多次执行相同的操作。
- 在每一步中，我们尝试较小的输入以使问题更小。
- 需要基本条件来停止递归，否则将发生无限循环。

A Mathematical Interpretation

- Let us consider a problem that a programmer has to determine the *sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n.*
- So the function simply looks like this,
  - ***approach(1) – Simply adding one by one***
    - $f(n) = 1 + 2 + 3 + \ldots\ldots + n$
- but there is another mathematical approach of representing this,
  - ***approach(2) – Recursive adding***
    - $f(n) = 1 \qquad n=1$
    - $f(n) = n + f(n-1) \qquad n>1$

There is a simple difference between approach (1) and approach(2) and that is in **approach(2)** the function " **f( )** " itself is being called inside the function, so this phenomenon is named recursion, and the function containing recursion is called recursive function, at the end, this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.

A Mathematical Interpretation

- 让我们考虑一个问题，程序员必须确定前n个自然数的总和，有几种方法可以做到这一点，但最简单的方法是简单地将1到n的数字相加。
- 所以函数只是看起来像这样
- ***approach(1) – Simply adding one by one***
  - ***f(n) = 1 + 2 + 3 +……..+ n***

- 但是还有另一种数学方法来表示这一点
- ***approach(2) – Recursive adding***
  - ***f(n) = 1 n=1***
  - ***f(n) = n + f(n-1) n>1***

方法（1）和方法（2）之间有一个简单的区别，那就是在方法（2）中函数“f（）”本身在函数内部被调用，所以这种现象被称为递归，而包含递归的函数称为递归函数，最后，这是程序员手中的一个很好的工具，可以以更简单有效的方式编写一些问题。

How are recursive functions stored in memory?

- **Recursion uses more memory**, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished.
- ***The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.***

●递归使用更多内存，因为递归函数会在每次递归调用时添加到堆栈中，并将值保留在那里，直到调用完成。

●递归函数使用 LIFO（后进先出）结构，就像堆栈数据结构一样。

What is the base condition in recursion?

- In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

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

在递归程序中，提供了基本情况的解决方案，
并以较小的问题表示较大问题的解决方案。

In the example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

How a particular problem is solved using recursion?

- The idea is to represent a problem in terms of one or more smaller problems and add one or more base conditions that stop the recursion.
- *For example, we compute factorial n if we know the factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.*

这个想法是用一个或多个较小的问题来表示问题，并添加一个或多个停止递归的基本条件。
例如，如果我们知道 （n-1） 的阶乘，我们计算阶乘 n。阶乘的基本情况为 n = 0。当 n = 0 时，我们返回 1。

Why Stack Overflow error occurs in recursion?

- If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

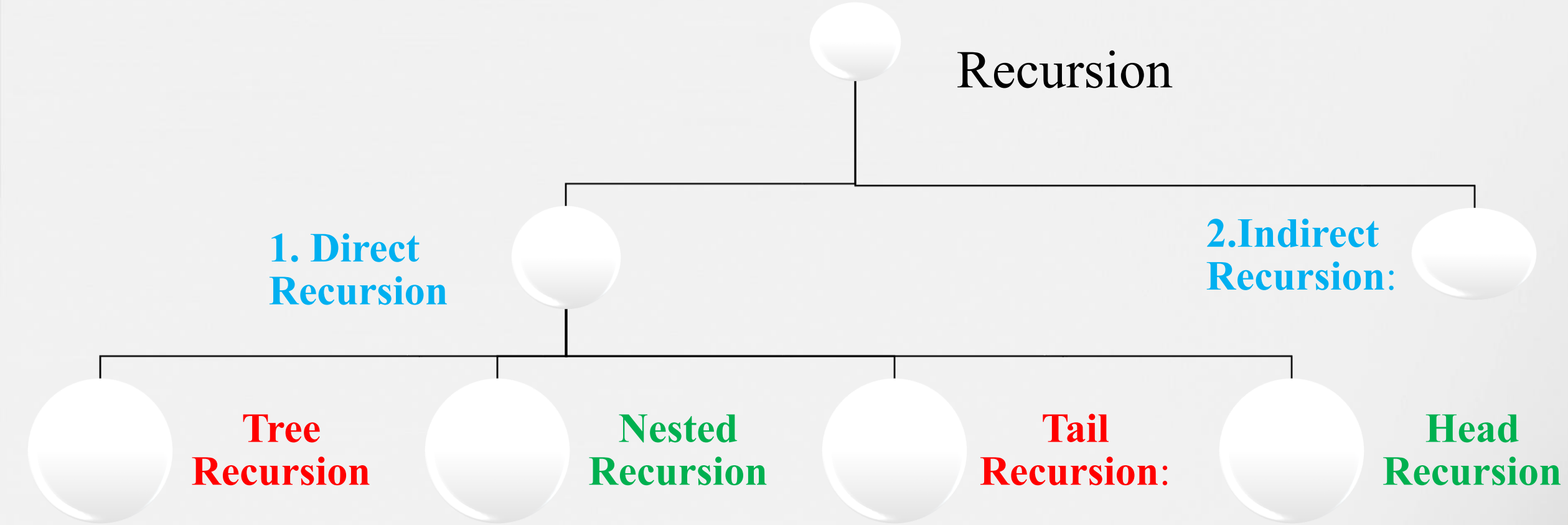
```
int fact(int n) {
 // wrong base case (it may cause stack overflow).
 if (n == 100)
return 1;
 else
return n*fact(n-1);
 }
```

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

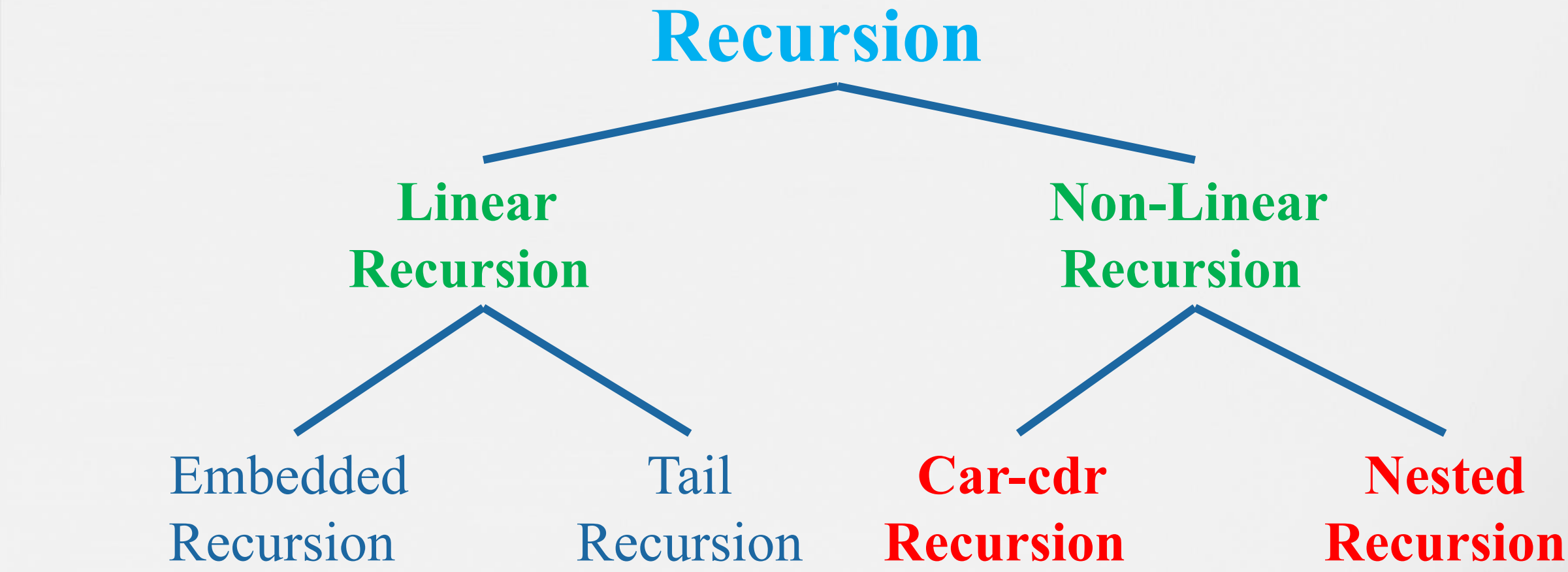
如果未达到或未定义基本情况，则可能会出现堆栈溢出问题。让我们举一个例子来理解这一点。

如果调用 fact（10），它将调用 fact（9）、fact（8）、fact（7） 等等，但数字永远不会达到 100。因此，未达到基本情况。如果堆栈上的这些函数耗尽了内存，则会导致堆栈溢出错误。

Recursion types?



Recursion



What is the difference between direct and indirect recursion?

- A function fun is called direct recursive if it calls the same function fun.

```
int fun(int z){
  fun(z-1);  //Recursive call
}
```

如果函数乐趣调用相同的函数乐趣，则称为直接递归。

calls itself again in its function body

- A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly.

```
int fun1(int z){
 int fun2(int y){
  fun2(z-1);
fun1(y-2)
}   }
```

如果函数 fun 调用另一个函数（例如 fun_new 并且fun_new直接或间接调用 fun），则称为间接递归。

the function fun1 explicitly calls fun2, which is invoking fun1 again

What is the difference between direct and indirect recursion?

- The difference between direct and indirect recursion

直接递归和间接递归的区别

```
// An example of direct recursion
void directRecFun() {
    // Some code....
 directRecFun();
  // Some code...
}
```

```
// An example of indirect recursion
void indirectRecFun1() {
     // Some code...
 indirectRecFun2();
    // Some code...
}
void indirectRecFun2() {
     // Some code...
 indirectRecFun1();
   // Some code...
}
```

1. Direct Recursion:

1. Direct Recursion:

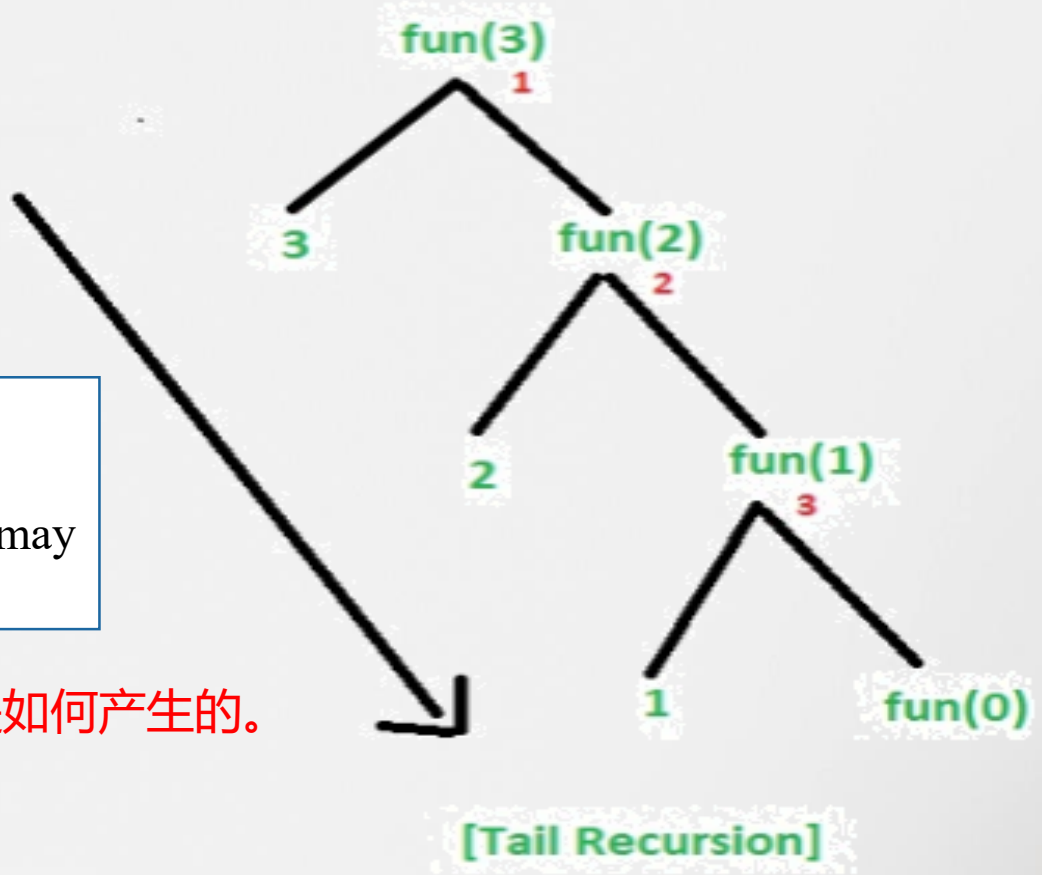
These can be further categorized into **four types**:

- **Tail Recursion**: If a recursive function calls itself and that recursive call is the last statement in the function then it's known as **Tail Recursion.**
- After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

- 1. 直接递归:
- 可以进一步分为四种类型:
- **尾部递归:**如果递归函数调用自身，而该递归调用是函数中的最后一条语句，则称为尾部递归。在调用递归函数之后，什么也不执行。函数在调用时必须处理或执行任何操作，而在返回时它什么也不做。

- Let's understand the example by **tracing tree of recursive function.** That is how the calls are made and how the outputs are produced.

Time Complexity For Tail Recursion : O(n)
Space Complexity For Tail Recursion : O(n)
Note: Time & Space Complexity is given for this specific example. It may vary for another example.

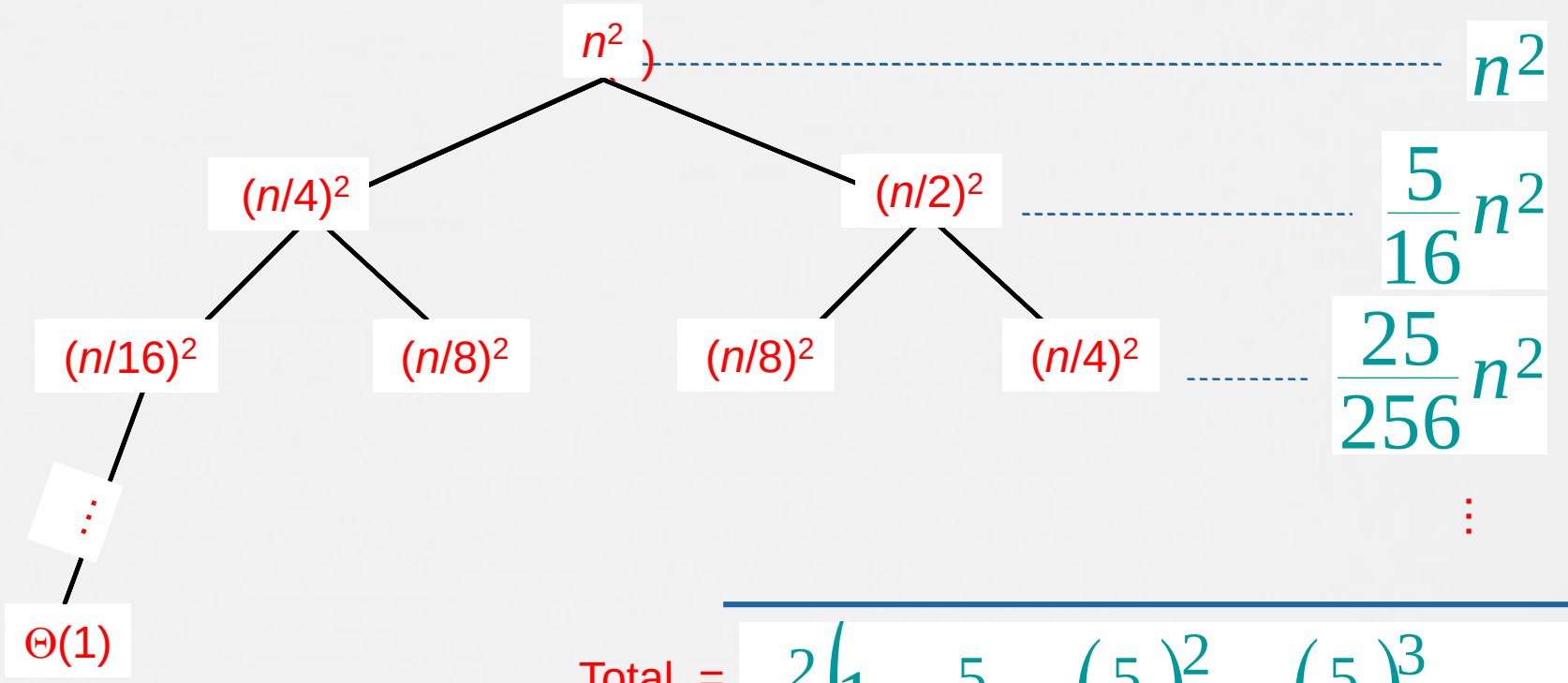


让我们通过跟踪递归函数树来理解这个例子。这就是调用和输出是如何产生的。
尾递归的时间复杂度:O(n)
尾递归的空间复杂度:O(n)

注意:对于这个特定的示例，给出了时间和空间复杂度。另一个例子可能会有所不同。

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$= \Theta(n^2)$ *geometric series*

Appendix: geometric series

$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for } |x| < 1$$

Compare

// Code Showing Tail Recursion

```
#include <iostream>
using namespace std;
 // Recursion function
void fun(int n)
{
    if (n > 0) {
        cout << na << " ";
         // Last statement in the function
        fun(n - 1);
    }
}
 // Driver Code
int main(){
    int x = 3;
    fun(x);
    return 0;
}
```

So it was seen that in case of loop the Space Complexity is O(1) so it was better to write code in loop instead of tail recursion in terms of Space Complexity which is more efficient than tail recursion.

由此可见，在循环的情况下，空间复杂度为O(1)，因此在空间复杂度方面，最好在循环中编写代码，而不是尾部递归，这比尾部递归更有效。

// Converting Tail Recursion into Loop

```
#include <iostream>
using namespace std;

void fun(int y)
{
    while (y > 0) {
        cout << y << " ";
        y--;
    }
}
 // Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Why space complexity is less in case of loop ?
为什么在循环情况下空间复杂度较低?

- **Why space complexity is less in the case of loop ?** In brief,when the program executes,the main memory divided into three parts. One part for code section, the second one is heap memory and another one is stack memory.
- **Remember that the program can directly access only the stack memory, it can't directly access the heap memory so we need the help of pointer to access the heap memory.**).

在解释这一点之前，我假设您已经熟悉了在程序执行期间如何将数据存储在主存中的知识。简单地说，当程序执行时，主存储器分为三个部分。一部分是代码部分，第二部分是堆内存，另一部分是堆栈内存。记住，程序只能直接访问栈内存，不能直接访问堆内存，因此需要借助指针来访问堆内存。

Why space complexity is less in case of loop ?
为什么在循环情况下空间复杂度较低?

- Let's now understand why space complexity is less in case of loop ?
  In case of loop when function "(void fun(int y))" executes there only one activation record created in stack memory(activation record created for only 'y' variable) so it takes only 'one' unit of memory inside stack so it's space complexity is O(1) but in case of recursive function every time it calls itself for each call a separate activation record created in stack.So if there's 'n' no of call then it takes 'n' unit of memory inside stack so it's space complexity is O(n).

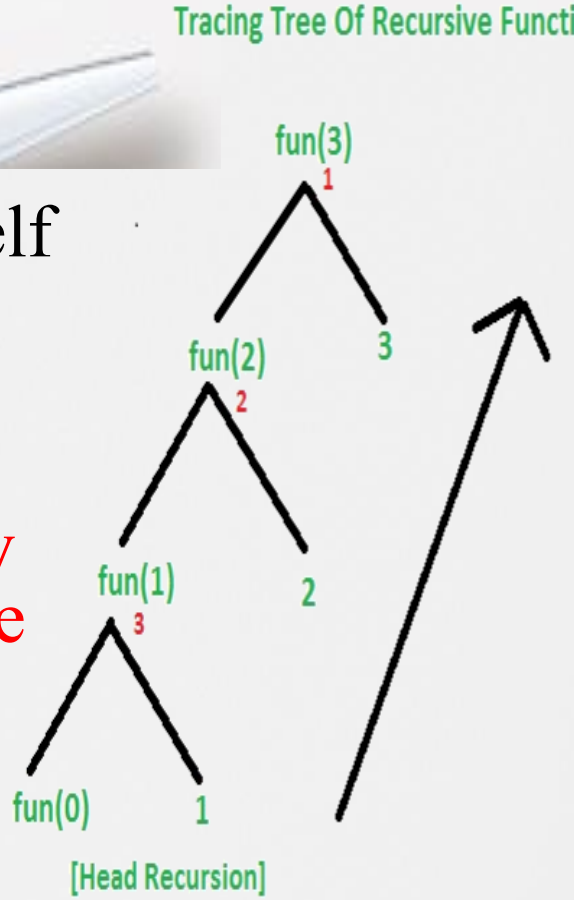
现在让我们理解为什么在循环的情况下空间复杂度更小?

在循环情况下，当函数" (void fun(int y)) "执行时，在堆栈内存中只创建了一个激活记录(仅为' y '变量创建的激活记录)，所以它在堆栈中只占用'一个'单元内存，所以它的空间复杂度是O(1)，但在递归函数中，每次调用它自己时，在堆栈中创建一个单独的激活记录。所以如果没有n个调用那么它在堆栈中占用n个单位的内存所以它的空间复杂度是O(n)

2. Head Recursion:

- **Head Recursion**: If a recursive function calling itself and that recursive call is the first statement in the function then it's known as **Head Recursion.**
- There's no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling and all operations are done at returning time.

- **Time Complexity For Head Recursion: O(n)**
  Space Complexity For Head Recursion: O(n)
- **Note:** Time & Space Complexity is given for this specific example. It may vary for another example.
  Note: Head recursion can't easily convert into loop as Tail Recursion but it can. Let's convert the above code into the loop.



Output: 1 2 3

*Digits in red showing that the order in which the calls are made and note that printing done at returning time. And it does nothing at calling time.

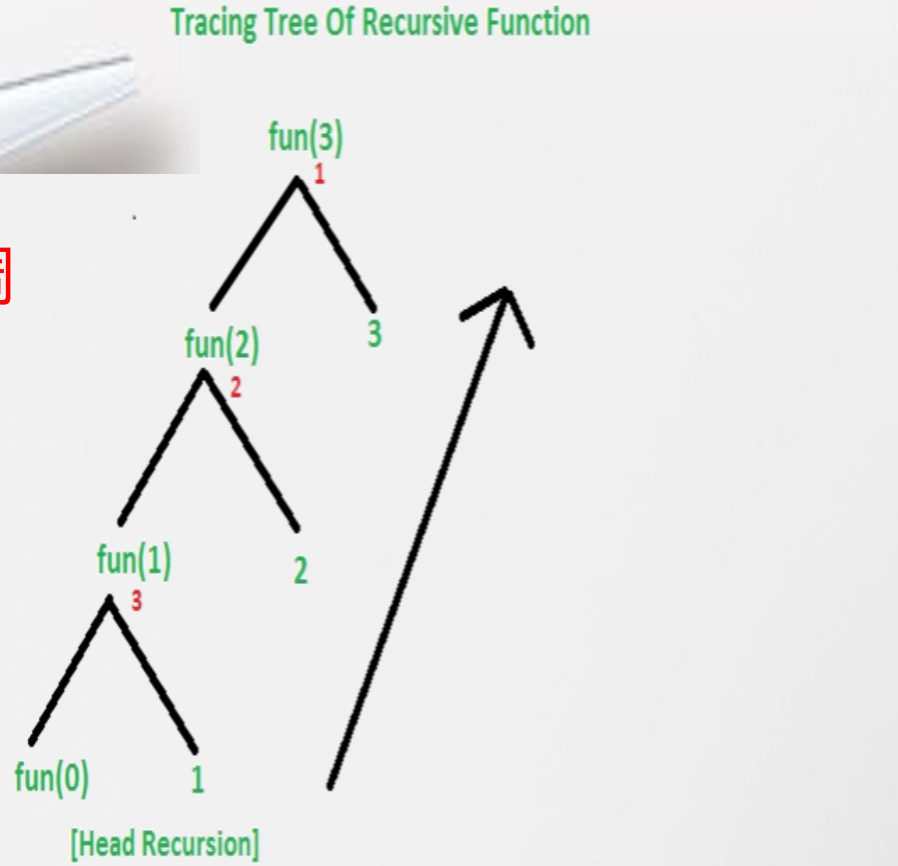
输出123

红色的数字表示调用的顺序，并说明在返回时打印完成。它在调用时什么也不做

2. Head Recursion:

- Head递归:如果一个递归函数调用自己，并且这个递归调用是函数中的第一个语句，那么它就被称为Head递归。
- 在调用之前没有语句，没有操作。函数在调用时不必处理或执行任何操作，所有操作都在返回时完成。

- **头递归的时间复杂度:O(n)**
- **头递归的空间复杂度:O(n)**
- **注意:对于这个特定的示例，给出了时间和空间复杂度。另一个例子可能会有所不同。**
- **注意:头递归不能像尾递归那样容易地转换为循环，但它可以。让我们将上面的代码转换到循环中。**



输出123
红色的数字表示调用的顺序，并说明在返回时打印完成。它在调用时什么也不做

Example on Head Recursion

```
// C++ program showing Head Recursion
#include <bits/stdc++.h>
using namespace std;

// Recursive function
void fun(int n)
{
    if (n > 0) {
        // First statement in the function
        fun(n - 1);
        cout << " "<< n;
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

```
// Converting Head Recursion into Loop
#include <iostream>
using namespace std;

// Recursive function
void fun(int n)
{
    int i = 1;
    while (i <= n) {
        cout <<" "<< i;
        i++;
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Tree Recursion:

- **Tree Recursion**: To understand **Tree Recursion** let's first understand **Linear Recursion**.
- If a recursive function calling itself for one time then it's known as **Linear Recursion**.
- Otherwise, if a recursive function calling itself for more than one time then it's known as **Tree Recursion**.

树递归:为了理解树递归，我们先来理解线性递归。如果一个递归函数调用自己一次，那么它被称为线性递归。否则，如果一个递归函数调用自己不止一次，那么它就被称为树递归。

- **Pseudo Code for linear recursion**

```
fun(n)
{
   // some code
   if(n>0)
   {
      fun(n-1); // Calling itself only once
   }
   // some code
}
```

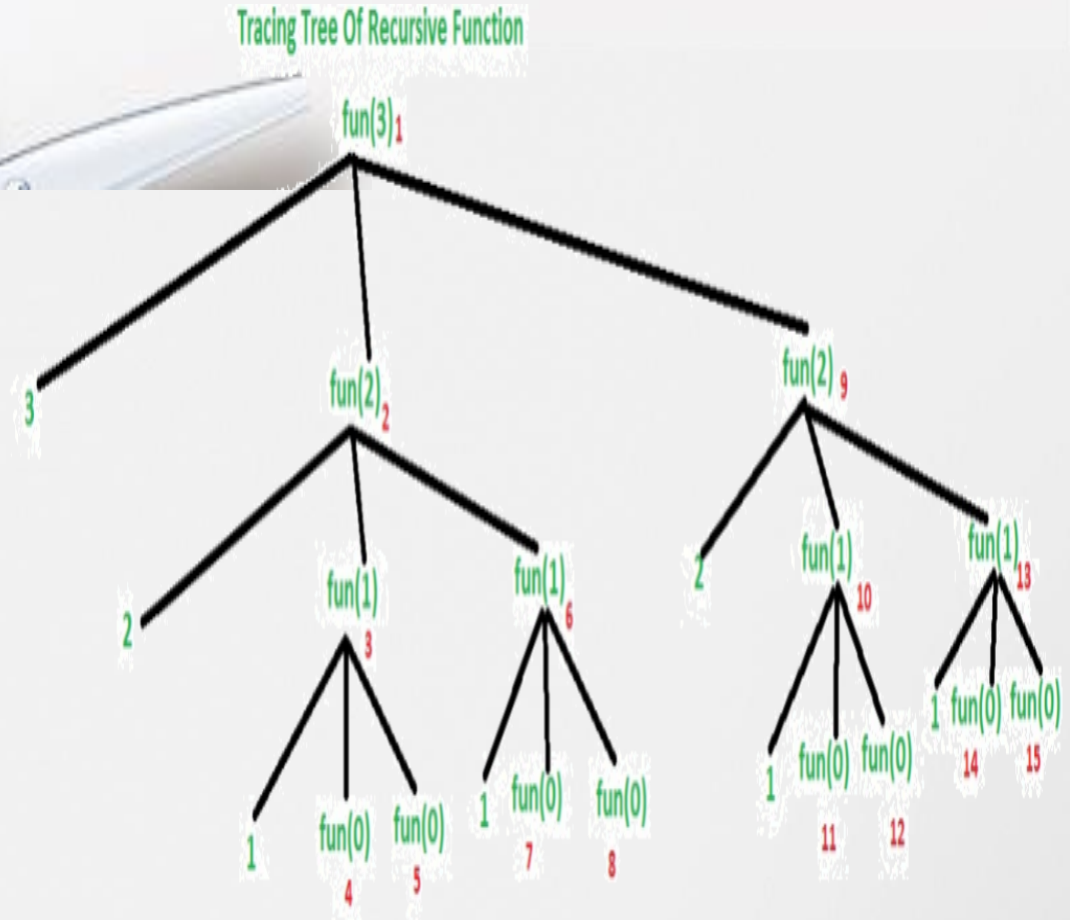
tracing tree of recursive function

- Let's understand the example by **tracing tree of recursive function.** That is how the calls are made and how the outputs are produced.

让我们通过跟踪递归函数树来理解这个例子。这就是调用和输出是如何产生的。

Time Complexity For Tree Recursion: O(2^n)
Space Complexity For Tree Recursion: O(n)
Note: Time & Space Complexity is given for this specific example. It may vary for another example.

树递归的时间复杂度:O(2^n)
树递归的空间复杂度:O(n)
注意:对于这个特定的示例，给出了时间和空间复杂度。另一个例子可能会有所不同。



Output: 3 2 1 1 2 1 1

*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

// C++ program to show Tree Recursion

```
// C++ program to show Tree Recursion
#include <iostream>
using namespace std;

// Recursive function
void fun(int n)
{
                if (n > 0)
                {
                 cout << " " << n;
                  // Calling once
                 fun(n - 1);
                // Calling twice
                 fun(n - 1);
                }
}

// Driver code
int main()
{
                fun(3);
                return 0;
}
```

Nested Recursion

- **Nested Recursion**: In this recursion, a recursive function will pass the parameter as a recursive call. That means **“recursion inside recursion”.** Let see the example to understand this recursion.

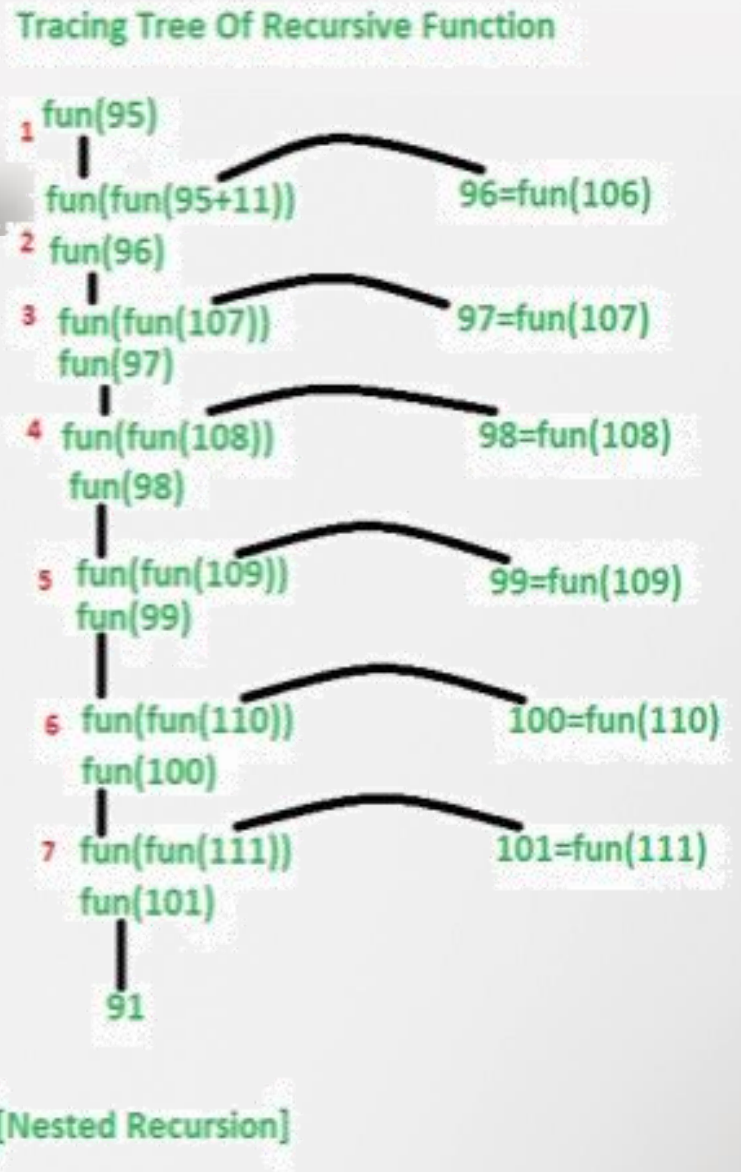
嵌套递归:在这种递归中，递归函数将作为递归调用传递参数。这意味着“递归中的递归”。让我们看看这个例子来理解这个递归。

```
// C++ program to show Nested Recursion
#include <iostream>
using namespace std;
int fun(int n){
        if (n > 100)
        return n - 10;
        // A recursive function passing parameter
        // as a recursive call or recursion inside
        // the recursion
        return fun(fun(n + 11));
}
// Driver code
int main(){
        int r;
        r = fun(95);
        cout << " " << r;
        return 0;
}
```

tracing tree of recursive function.

- Let's understand the example by **tracing tree of recursive function.**
- That is how the calls are made and how the outputs are produced.

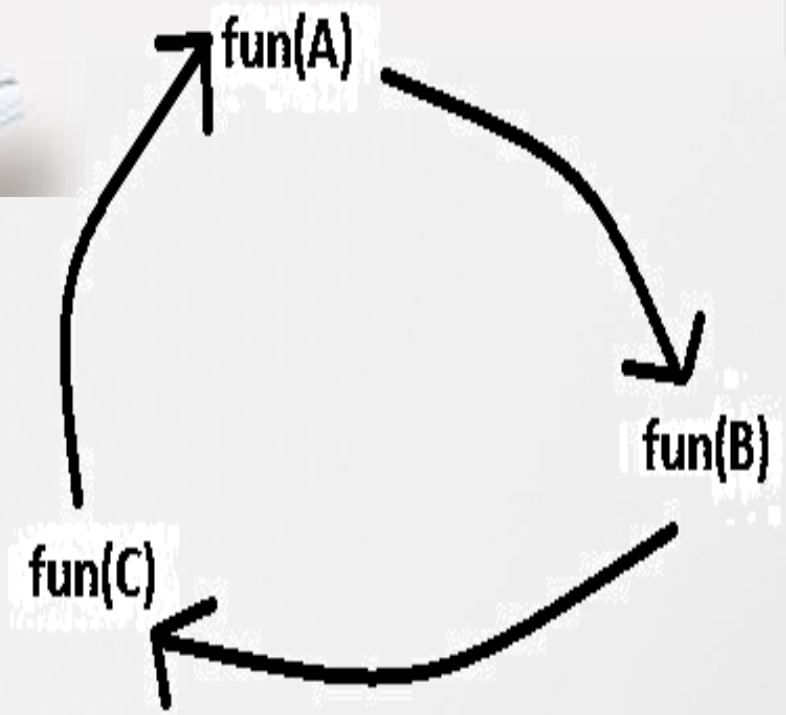
让我们通过跟踪递归函数树来理解这个例子。这就是调用和输出是如何产生的。



红色数字表示童话的顺序

2. Indirect Recursion:

- **2. Indirect Recursion**: In this recursion, there may be more than one functions and they are calling one another in a circular manner.



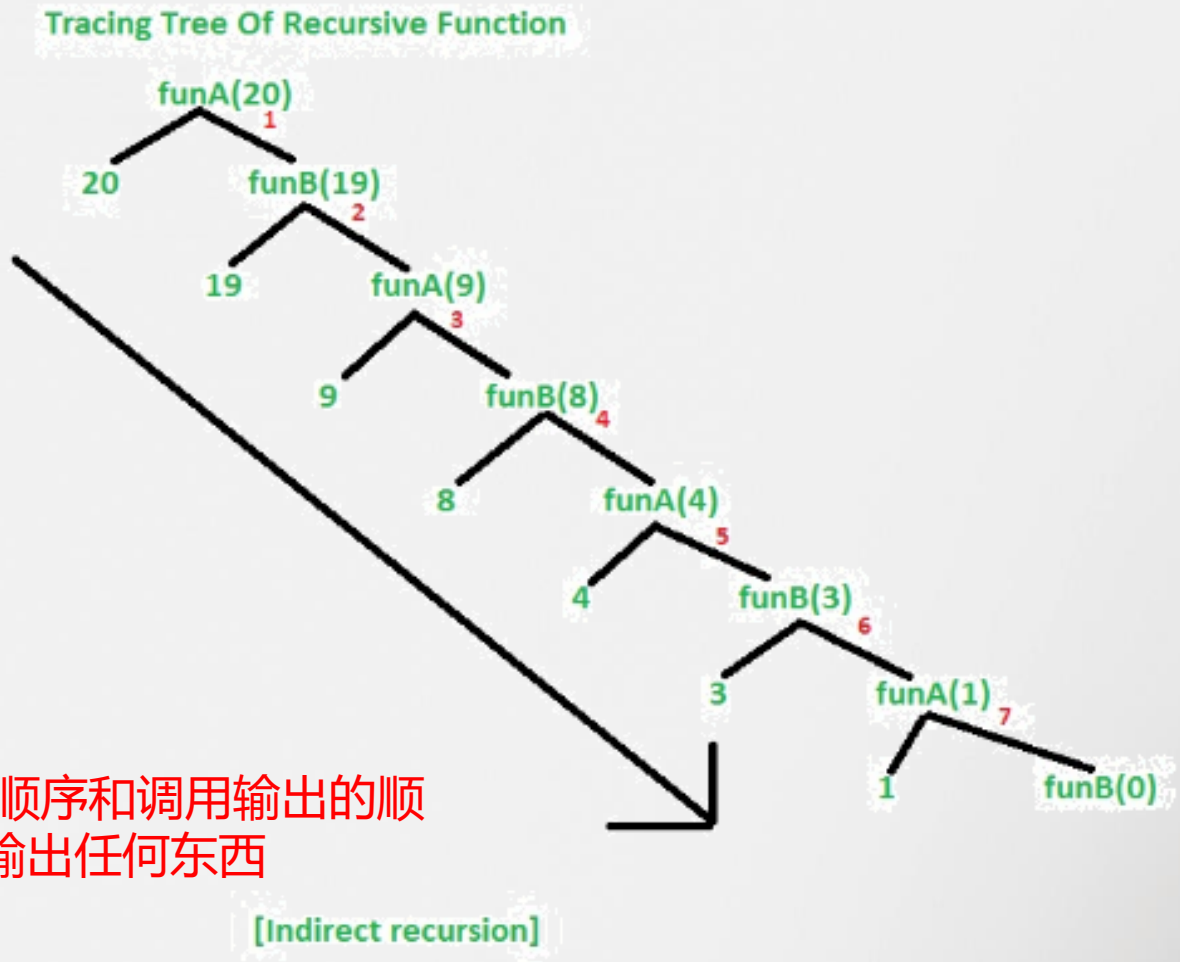
2. 间接递归:在这种递归中，可能有多个函数，它们以循环的方式相互调用。

- **From the above diagram fun(A) is calling for fun(B), fun(B) is calling for fun(C) and fun(C) is calling for fun(A) and thus it makes a cycle.**

从上面的图表来看，fun(A)呼唤fun(B)，fun(B)呼唤fun(C)，fun(C)呼唤fun(A)，这样就形成了一个循环。

tracing tree of recursive function

- Let's understand the example by **tracing tree of recursive function.** That is how the calls are made and how the outputs are produced.



让我们通过跟踪递归函数树来理解这个例子。这就是调用和输出是如何产生的

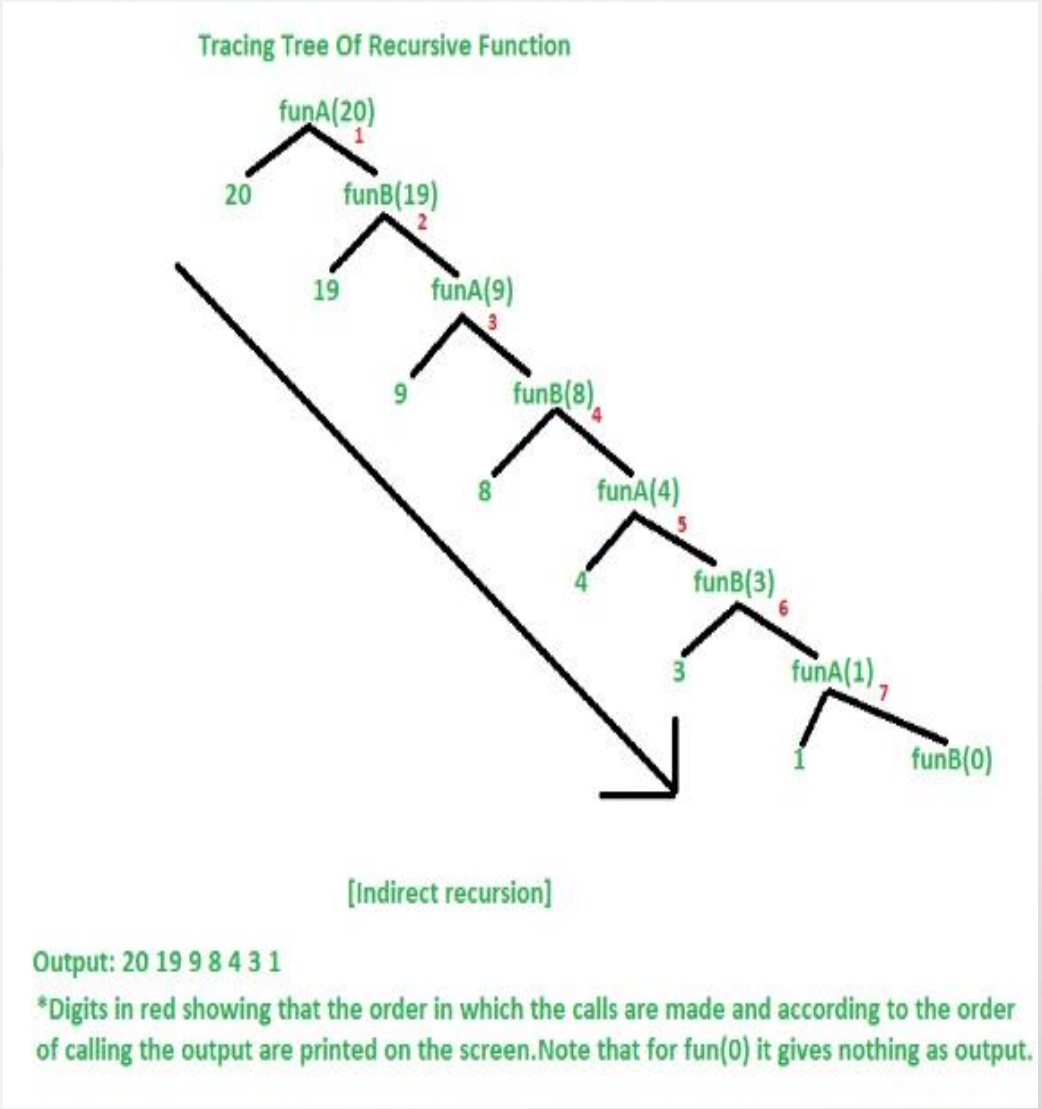
输出：20 19 9 8 4 3 1
屏幕上显示的红色数字表示调用的顺序和调用输出的顺序。注意，为了好玩（D）他没有输出任何东西

Output: 20 19 9 8 4 3 1

*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

// C++ program to show Indirect Recursion

```
// C++ program to show Indirect Recursion
#include <iostream>
using namespace std;
void funB(int n);
void funA(int n)
{
 if (n > 0) {
                cout <<" "<< n;
                 // fun(A) is calling fun(B)
                 funB(n - 1);
                }
}
void funB(int n){
if (n > 1) {
cout <<" "<< n;
// fun(B) is calling fun(A)
funA(n / 2);
}
}
// Driver code
int main()
{
                funA(20);
                return 0;
}
```



Steps to Solve Recurrence Relations Using Recursion Tree Method

- **Step-01:**Draw a recursion tree based on the given recurrence relation.
- 根据给定的递归关系绘制递归树
- **Step-02:**
  - Determine- 分析
  - Cost of each level 每层数量
  - Total number of levels in the recursion tree 递归树的总层数
  - Number of nodes in the last level 最后一层的节点数
  - Cost of the last level 最后一层的数量
- **Step-03:**
- Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.
- Following problems clearly illustrates how to apply these steps.
- 将递归树所下面的问题清楚地说明了如何应用这些步骤。
- 有层的数量相加，用渐近表示法将得到的表达式简化

Example on board_1

$$T(n) = \begin{cases} 1 & n = 1 \\ 2\,T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Solve the following recurrence using the Recurrence Tree Method.

Assumption: We assume that n is exact power of 2.

$$x^{\log_y n} \Longrightarrow n^{\log_y x}$$

$$x^0 + x^1 + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1} \qquad for\ x \neq 1$$

$$x^0 + x^1 + x^2 + \cdots = \frac{1}{1 - x} \qquad for\ |x| < 1$$

Problem-01:

- **Problem-01:**
- Solve the following recurrence relation using recursion tree method
- $T(n) = 2T(n/2) + n$

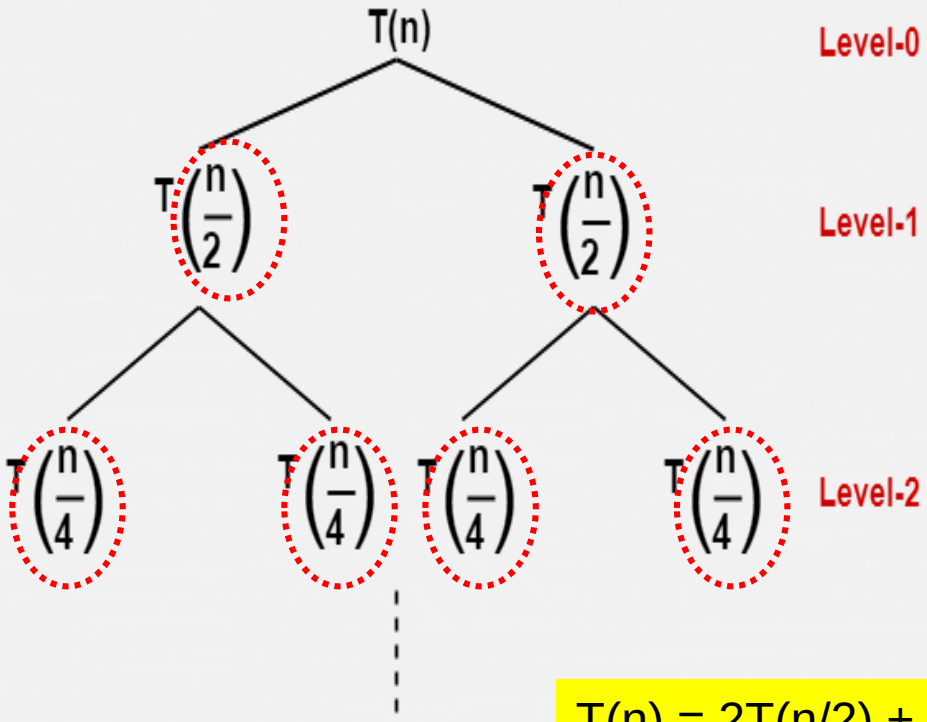
用递归树法求解下列递归关系

- **Step-01:**
- Draw a recursion tree based on the given recurrence relation.
- The given recurrence relation shows-
  - A problem of size n will get divided into 2 sub-problems of size n/2.
  - Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
  - At the bottom most layer, the size of sub-problems will reduce to 1.

- **Step-01:**
- 根据给定的递归关系绘制递归树所给递归关系式为：
- 一个大小为n的问题会被分成两个大小为n/2的子问题
  - 然后，每个大小为n/2的子问题将被分成 两个大小为n/4的子问题，以此类推。
  - 在最底部的一层，子问题的大小将减少 到1

Problem-01:

- This is illustrated through following recursion tree-
- 通过下面的递归树来说明这一点



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

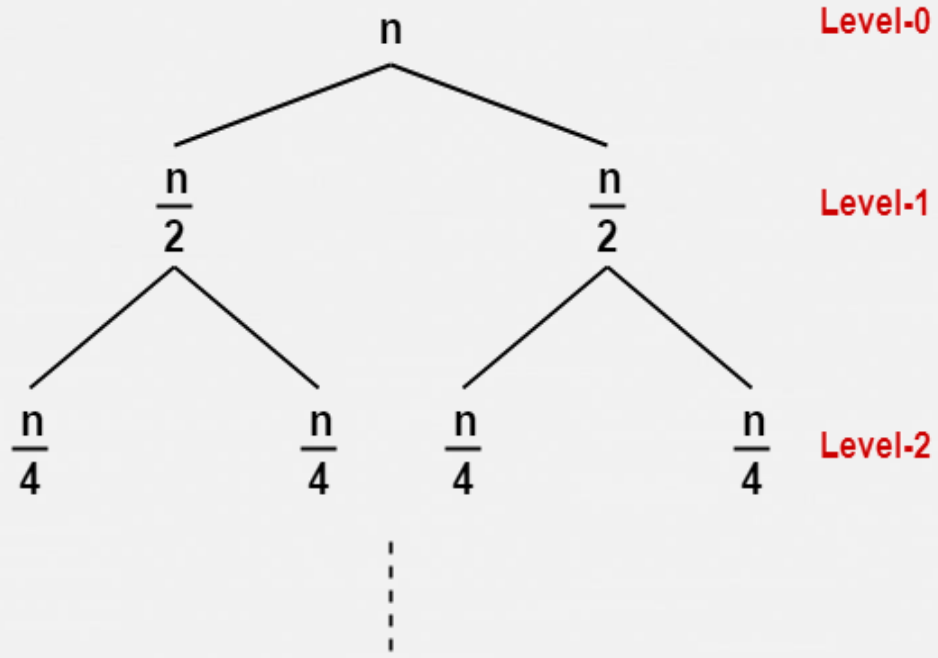
The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/2 into its 2 sub-problems and then combining its solution is n/2 and so on.

- 所给递归关系式为：
- 将一个大小为n的问题分成两个子问题，然后将其解合并的代价是n。
- 将一个大小为n/2的问题分解为两个子问题，然后将其解组合为n/2，以此类推

Problem-01:

- This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-
- 这可以通过下面的递归树来说明，其中每个节点代表一个数量



- **Step-02:**
- Determine cost of each level-
  - Cost of level-0 = n
  - Cost of level-1 = n/2 + n/2 = n
  - Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n and so on.

确定每一级的数量

level-0的数量 = n

level-1的数量 = n/2 + n/2 = n

level-2的数量 = n/4 + n/4 + n/4 + n/4 = n and so on.

Problem-01:

- **Step-03:**

Determine total number of levels in the recursion tree-

- Size of sub-problem at level-0 = $n/2^0$
- Size of sub-problem at level-1 = $n/2^1$
- Size of sub-problem at level-2 = $n/2^2$

确定递归树中的总级别数

- level-0子问题的大小 = $n/2^0$
- level-1子问题的大小 = $n/2^1$
- level-2子问题的大小 = $n/2^2$

- Continuing in similar manner, we have-
- Size of sub-problem at level-i = n/2i
- Suppose at level-x (last level), size of sub-problem becomes 1.
- Then-
  - n / 2x = 1
  - 2x = n
- Taking log on both sides, we get-
  - xlog2 = logn
  - x = log2n
- ∴ Total number of levels in the recursion tree = log2n + 1

- 继续以同样的方式，我们有-
- 子问题level-i 的大小 = n/2i
- 假设在第x层(最后一层)，子问题的大小为1。然后，
  - n / 2x = 1
  - 2x = n
- 两边同时取对数，得到-
  - xlog2 = logn
  - x = log2n
- ∴ 递归树的总层数 = log2n + 1

Problem-01:

- **Step-04:**
  - Determine number of nodes in the last level-
  - Level-0 has 2^0 nodes i.e. 1 node
  - Level-1 has 2^1 nodes i.e. 2 nodes
  - Level-2 has 2^2 nodes i.e. 4 nodes
  - Continuing in similar manner, we have-
  - Level-$\log_2 n$ has $2^{\log_2 n}$ nodes i.e. n nodes

确定最后一层的节点数
- Level-0有20个节点，即1个节点
- Level-1有21个节点，即2个节点
- Level-2有22个节点，即4个节点
- 继续以同样的方式，我们有-
- Level-log2n有2log2n个节点，即n个节点

- **Step-05:**
  - Determine cost of last level-
  - Cost of last level = n x T(1) = θ(n)

- 确定上一级的数量
- 上一层数量= n x T(1) = θ(n)

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

将递归树的所有级别的代价相加，并将得到的表达式简化为渐近符号-

$$T(n) = \underbrace{\{ n + n + n + \ldots\ldots \}}_{\text{For } \log_2 n \text{ levels}} + \theta(n)$$

Problem-01:

$T(n) = \underbrace{\{ n + n + n + \ldots\ldots \}}_{\text{For } \log_2 n \text{ levels}} + \theta(n)$

$= n \times \log_2 n + \theta(n)$

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

$= \boldsymbol{\theta(n\log_2 n)}$

Recursion Tree Method: Example 1

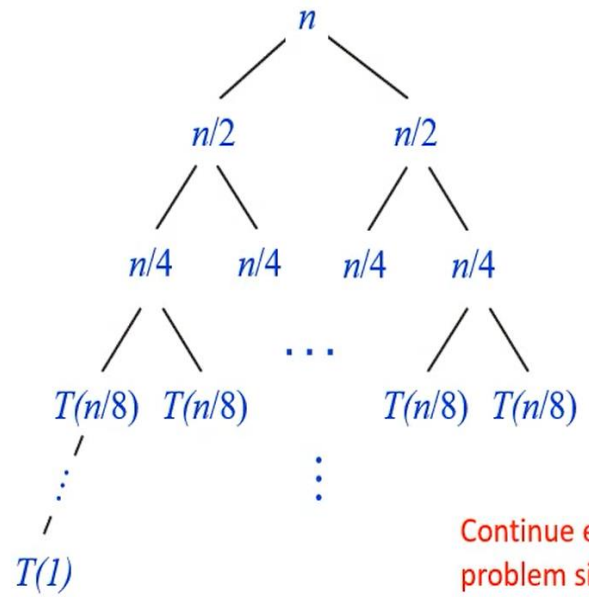
$T(n) = 2\,T\left(\frac{n}{2}\right) + n$

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

$T\left(\frac{n}{2^2}\right) = 2.T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$

$T\left(\frac{n}{2^{k-1}}\right) = 2.T\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-1}}$

$T\left(\frac{n}{2^k}\right) = T(1)$



Continue expanding until the problem size reduces to 1.

Total Cost = Cost of Leaf Nodes + Cost of Internal Nodes

Total Cost = (cost of leaf node x total leaf nodes) + (sum of costs at each level of internal nodes)

Total Cost = $L_c + I_c$

Example on board_2

$$T(n) = \begin{cases} 1 & n = 1 \\ 2\,T\left(\frac{n}{2}\right) + n^2 & n > 1 \end{cases}$$

Solve the following recurrence using the Recurrence Tree Method.

Assumption: We assume that n is exact power of 2.

Example on board_3

$$T(n) = \begin{cases} 1 & n = 1 \\ 3\,T\left(\frac{n}{4}\right) + n^2 & n > 1 \end{cases}$$

Solve the following recurrence using the Recurrence Tree Method.

Assumption: We assume that n is exact power of 4.

Example on board_4

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2 & n > 1 \end{cases}$$

Solve the following recurrence using the Recurrence Tree Method.

Example on board_5

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n & n > 1 \end{cases}$$

Solve the following recurrence using the Recurrence Tree Method.

Problem-02:

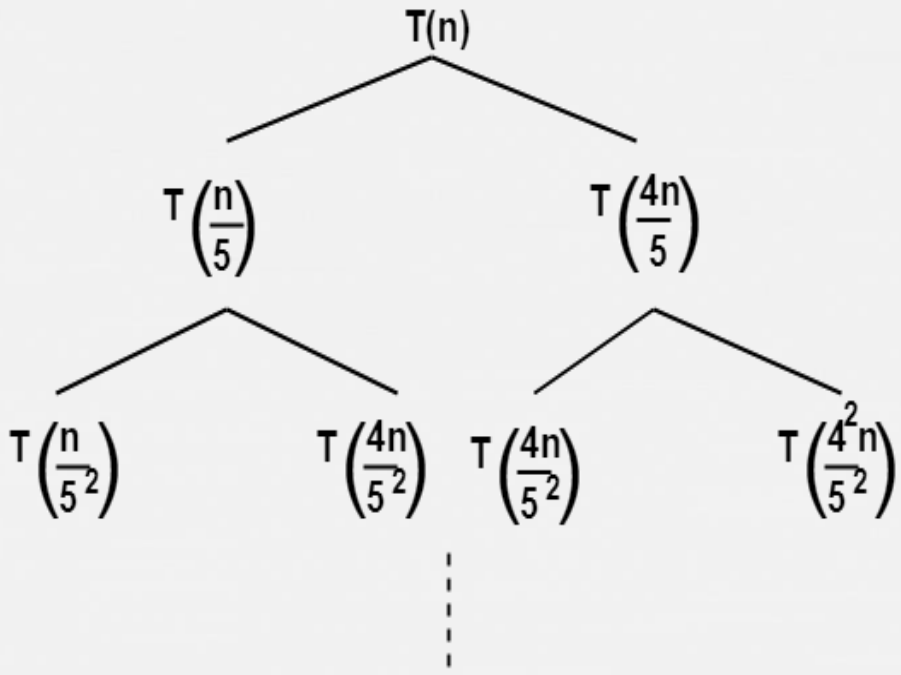
- **Problem-02:**
- **Solve the following recurrence relation using recursion tree method-**
- 用递归树法求解以下递归关系-
- **$T(n) = T(n/5) + T(4n/5) + n$**

- **Step-01:**
- Draw a recursion tree based on the given recurrence relation.
- The given recurrence relation shows-
- A problem of size n will get divided into 2 sub-problems- one of size n/5 and another of size 4n/5.
- Then, sub-problem of size n/5 will get divided into 2 sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
- On the other side, sub-problem of size 4n/5 will get divided into 2 sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$ and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.

- **Step-01:**
- 根据给定的递归关系绘制递归树。
- 所给递归关系式为-
- 一个大小为n的问题将被分为两个子问题——一个大小为n/5，另一个大小为4n/5。
- 然后，大小为n/5的子问题将被分为两个子问题——一个大小为n/52，另一个大小为4n/52。
- 另一方面，大小为4n/5的子问题将被分为两个子问题——一个大小为4n/52，另一个大小为42n/52，以此类推。
- 在最底部的一层，子问题的大小将减少到1。

Problem-02:

- This is illustrated through following recursion tree-
- 通过下面的递归树-来说明这一点



The given recurrence relation shows-

•The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.

•The cost of dividing a problem of size n/5 into its 2 sub-problems and then combining its solution is n/5.

•The cost of dividing a problem of size 4n/5 into its 2 sub-problems and then combining its solution is 4n/5 and so on.

•所给递归关系式为-

•将一个大小为n的问题分成两个子问题，然后将其解合并的代价是n。

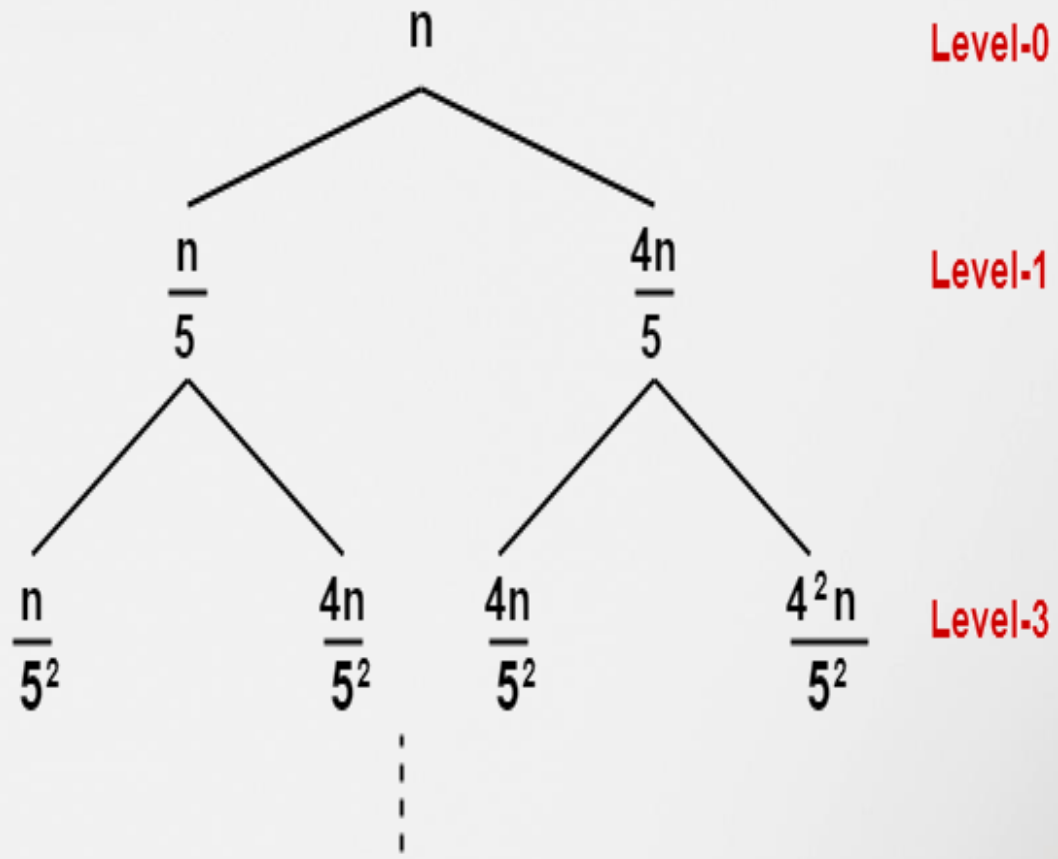
•将一个大小为n/5的问题分解为2个子问题，然后将其解合并的代价是n/5。

•将一个大小为4n/5的问题分成两个子问题，然后将其解组合为4n/5，以此类推。

Problem-02:

- This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-

这可以通过下面的递归树来说明，其中每个节点表示相应子问题的代价-



Problem-02:

- **Step-02:**
- Determine cost of each level-
- Cost of level-0 = n
- Cost of level-1 = $n/5 + 4n/5 = n$
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

- **Step-02:**
- 确定每一级的数量
- 0级的数量= n
- 第一级的数量= $n/5 + 4n/5 = n$
- 二级的数量= $n/52 + 4n/52 + 4n/52 + 42n/52 = n$

Step-03:

Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-

- Size of sub-problem at level-0 = $(4/5)^0n$
- Size of sub-problem at level-1 =$(4/5)^1n$
- Size of sub-problem at level-2 =$(4/5)^2n$

确定递归树中的总级别数。我们将考虑最右边的子树，因为它向下到最深的层-

- 第0级子问题的大小= (4/5)0n
- 第1级子问题的大小=(4/5)1n
- 第2级子问题的大小=(4/5)2n
- 继续以同样的方式，我们有-
- 第i级子问题的大小= (4/5)in

Problem-02:

Step-03:

Continuing in similar manner, we have-

Size of sub-problem at level-i = $(4/5)^i n$

Suppose at level-x (last level), size of sub-problem becomes 1.

Then-

$(4/5)^x n = 1$

$(4/5)^x = 1/n$

Taking log on both sides, we get-

$x\log(4/5) = \log(1/n)$

$x = \log_{5/4} n$

∴ Total number of levels in the recursion tree = $\log_{5/4} n + 1$

Step-03:

假设在第x层(最后一层)，子问题的大小为1。

然后，

- $(4/5)xn = 1$
- $(4/5)x = 1/n$

两边同时取对数，得到-

- $X\log(4/5) = \log(1/n)$
- $X = \log 5/4 n$

∴递归树的总级别数= $\log 5/4 n + 1$

Problem-02:

- **Step-04:**
- Determine number of nodes in the last level-
  - Level-0 has 2^0 nodes i.e. 1 node
  - Level-1 has 2^1 nodes i.e. 2 nodes
  - Level-2 has 2^2 nodes i.e. 4 nodes
- Continuing in similar manner, we have-
- Level-$\log_{5/4}n$ has $2^{\log_{5/4}n}$ nodes

- **Step-05:**
- 确定上一级的成本
- 上一层成本=

$2\log_{5/4}n \times T(1) = \theta(2\log_{5/4}n) = \theta(n\log_{5/4}2)$

- **Step-04:**
- 确定最后一层的节点数
  - Level-0有20个节点，即1个节点
  - Level-1有21个节点，即2个节点
  - Level-2有22个节点，即4个节点
- 继续以同样的方式，我们有-
- Level-log5/4n有2log5/4n个节点

- **Step-05:**
- Determine cost of last level-
- Cost of last level = $2^{\log_{5/4}n} \times T(1)$ $= \theta(2^{\log_{5/4}n}) = \theta(n^{\log_{5/4}2})$

Problem-02:

- **Step-06:**
- Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

将递归树的所有级别的代价相加，并将得到的表达式简化为渐近符号-

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_{5/4} n \text{ levels}} + \theta(n^{\log_{5/4} 2})$$

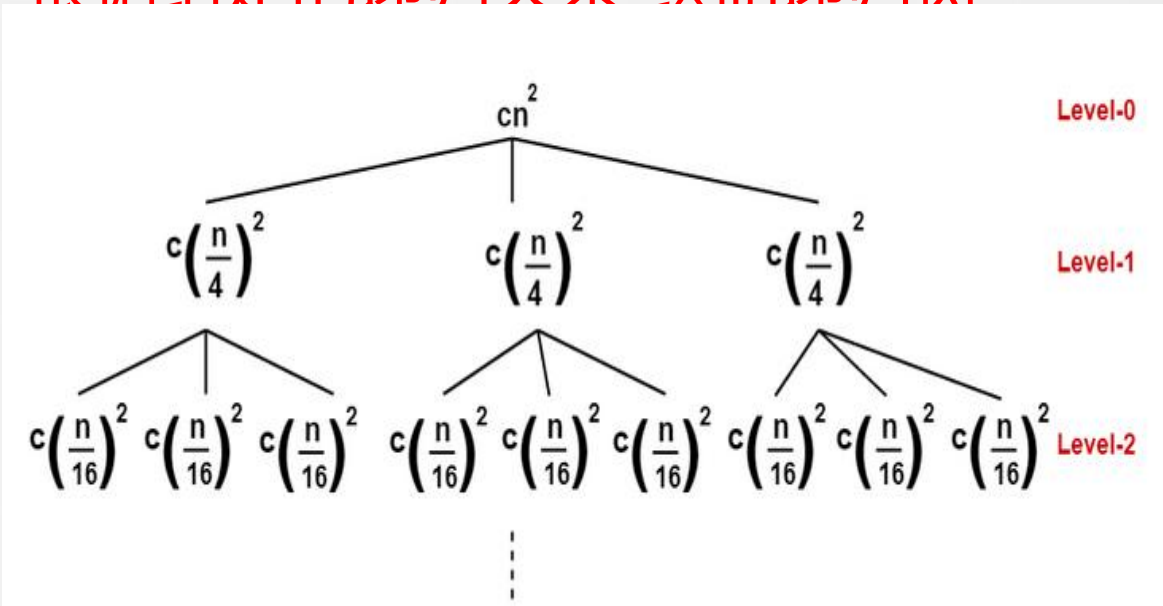
$$= n\log_{5/4} n + \theta(n^{\log_{5/4} 2})$$

$$= \boldsymbol{\theta(n\log_{5/4} n)}$$

Problem-03:

- Solve the following recurrence relation using recursion tree method-
- 用递归树法求解以下递归关系-
- $T(n) = 3T(n/4) + cn^2$

- **Step-01:**
- Draw a recursion tree based on the given recurrence relation-
- 根据给定的递归关系-绘制递归树



(Here, we have directly drawn a recursion tree representing the cost of sub problems)
(这里，我们直接画了一个递归树，表示子问题的代价)

Problem-03:

- **Step-02:**
  - Determine cost of each level-
  - Cost of level-0 = cn^2
  - Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
  - Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

- **Step-02:**
  - 确定每一级的成本
  - 0级的花费= cn2
  - 一级代价= c(n/4)2 + c(n/4)2 + c(n/4)2 = (3/16)cn2
  - 二级成本= c(n/16)2 × 9 = (9/162)cn2

Problem-03:

Step-03:

Determine total number of levels in the recursion tree-

Size of sub-problem at level-0 = $n/4^0$

Size of sub-problem at level-1 = $n/4^1$

Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have-

Size of sub-problem at level-i = $n/4^i$

Suppose at level-x (last level), size of sub-problem becomes 1.

Then-

$$n/4^x = 1$$
$$4^x = n$$

Taking log on both sides, we get-

$$x\log 4 = \log n$$
$$x = \log_4 n$$

∴ Total number of levels in the recursion tree = $\log_4 n + 1$

Step-03:

确定递归树中的总级别数

第0级子问题的大小= n/40

第一级子问题的大小= n/41

第2级子问题的大小= n/42

继续以同样的方式，我们有-

第i级子问题的大小= n/4i

假设在第x层(最后一层)，子问题的大小为1。然后，

N /4x = 1

4x = n

两边同时取对数，得到-

Xlog4 = logn

X = log4n

∴递归树的总级别数= log4n + 1

Problem-03:

- **Step-04:**
- Determine number of nodes in the last level-
- Level-0 has 3^0 nodes i.e. 1 node
- Level-1 has 3^1 nodes i.e. 3 nodes
- Level-2 has 3^2 nodes i.e. 9 nodes
- Continuing in similar manner, we have-
- Level-$\log_4 n$ has $3^{\log_4 n}$ nodes i.e. $n^{\log_4 3}$ nodes

确定最后一层的节点数

- Level-0有30个节点，即1个节点
- Level-1有31个节点，即3个节点
- Level-2有32个节点，即9个节点
- 继续以同样的方式，我们有-
- Level-log4n有3log4n个节点，即nlog43个节点

- **Step-05:**
- Determine cost of last level-
- Cost of last level = $n^{\log_4 3} \times T(1) = \theta(n^{\log_4 3})$

确定上一级的数量

上一层数量= nlog43 × T(1) = θ(nlog43)

Problem-03:

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\left\{ cn^2 + \frac{3\,cn^2}{16} + \frac{9\,cn^2}{(16)^2} + \ldots\ldots\ldots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta\left(n^{\log_4 3}\right)$$

$= cn^2 \{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \} + \theta(n^{\log_4 3})$

Now, $\{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \}$ forms an infinite Geometric progression.

On solving, we get-

$= (16/13)cn^2 \{ 1 - (3/16)^{\log_4 n} \} + \theta(n^{\log_4 3})$

$= (16/13)cn^2 - (16/13)cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$

$= \mathbf{O(n^2)}$

Reference

- AD book
- https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/

江西理工大学
Jiangxi University of Science and Technology
信息工程学院
School of information engineering

高级算法分析与设计

Advanced Algorithm Analysis and Design

THANK YOU

"The beauty of research is that you never know where it's going to lead."

研究的美妙之处在于你永远不知道它会通向何方

RICHARD ROBERTS
Nobel Prize in Physiology or Medicine 1993

"BE HUMBLE. BE HUNGRY.
AND ALWAYS BE THE
HARDEST WORKER
IN THE ROOM."