算法设计与分析

Lecture 4: Recursion (递归)

曹刘娟

厦门大学信息学院计算机系

caoliujuan@xmu.edu.cn

Recursion

递归基本思想

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that would be difficult to solve in other ways. 递归通过生成一系列的更容易的问题, 并逐个求解，拼接答案来求解原有问题。
- Recursion splits an problem instance into one or more simpler instances of the same problem.

将一个问题切分成为一个或者多个更小规模的同样问题。（同样方法解决）

递推的思维方式

- 通常我们的**思维方式是递推的**，对世界的认识是由近即远，从少到多，从小到大
  - 数数，从1，2，3到100
  - 数学归纳法
- 这种递推的方式**限制**了我们的想象力和大局观，当需要思维触达那些远离我们生活经验的地方时，我们可能会出现理解障碍。

递归是一种逆向思考的方式

- 计算机是用来设计处理规模很大的问题，因此需要和常人不同的思维方式来解决问题。**递归这种思维方式是要自顶向下，先全局后局部，从大到小**
- 能否掌握递归这种思维方法是学好计算机的关键

Image source: https://giphy.com/explore/infinite-recursion   https://baijiahao.baidu.com/s?id=1676102705729489021&wfr=spider&for=pc

Design a Recursive Algorithm 设计递归算法

- Base case （递归出口）: There must be at least one case, for a small value of n, that can be solved directly.规模n小到某值，可直接求解
- Recursive case （递归情况）: A problem instance of a given size n can be split into one or more smaller instances of the same problem.
- Steps:（基本设计步骤）
  - Recognize the base case and provide a quick solution to it. 确定递归出口，并且提供快速解法
  - Devise a recursion to split the instance into smaller instances of itself, while making progress toward the base case.设计一个策略，将大问题递归的分解为小问题，直至基本情况问题，并且要求子问题的循环不变量为真
  - Combine the solutions of the smaller problems in such a way as to solve the larger problem.将小问题的解法合并起来，成为大问题的解法

Design a Recursive Algorithm

Questions when using recursive solution:（需要注意的几个问题）

- How to define the problem in terms of a smaller problem of the same type? 如何将原问题定义成一系列同样类型但是更小规模的问题
- How does each recursive call diminish the size of the problem? 如何递归的降低原问题的规模
- What instance of the problem can serve as the base case? 子问题要满足什么条件才能直接求解
- As the problem size diminishes, will you reach this base case? 当问题逐渐变小时，递归出口能达到么?

Why Use Recursion?

- Advantages
  - Interesting conceptual framework (good recursion algorithm is art).
  - Intuitive solutions to difficult problems. 对于一些难的问题，可以用比较优雅的解法来解决
- But, disadvantages...
  - More memory & time. 消耗更多的内存以及时间
  - Different way of thinking! 与递推不一样的思维方式

Correctness of Recursive Algorithm

Correctness proof of recursion is similar to induction. 用数学归纳法来**证明递归算法的正确性**

- Base case: Verify that the base case is recognized and solved correctly. 检验递归出口能正确，初始情况下，循环不变量为真

- Induction step: Verify that if all smaller problems are solved correctly, then the original problem is also solved correctly.

  所有子问题都能正确解决，原问题也能正确解决

Example 1

Consider the function $f(n)$ which calculates 2 to the power of n, namely $f(n) = 2^n$. 计算函数的幂

This can be expressed as:

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2 \times f(n-1) & \text{otherwise.} \end{cases}$$

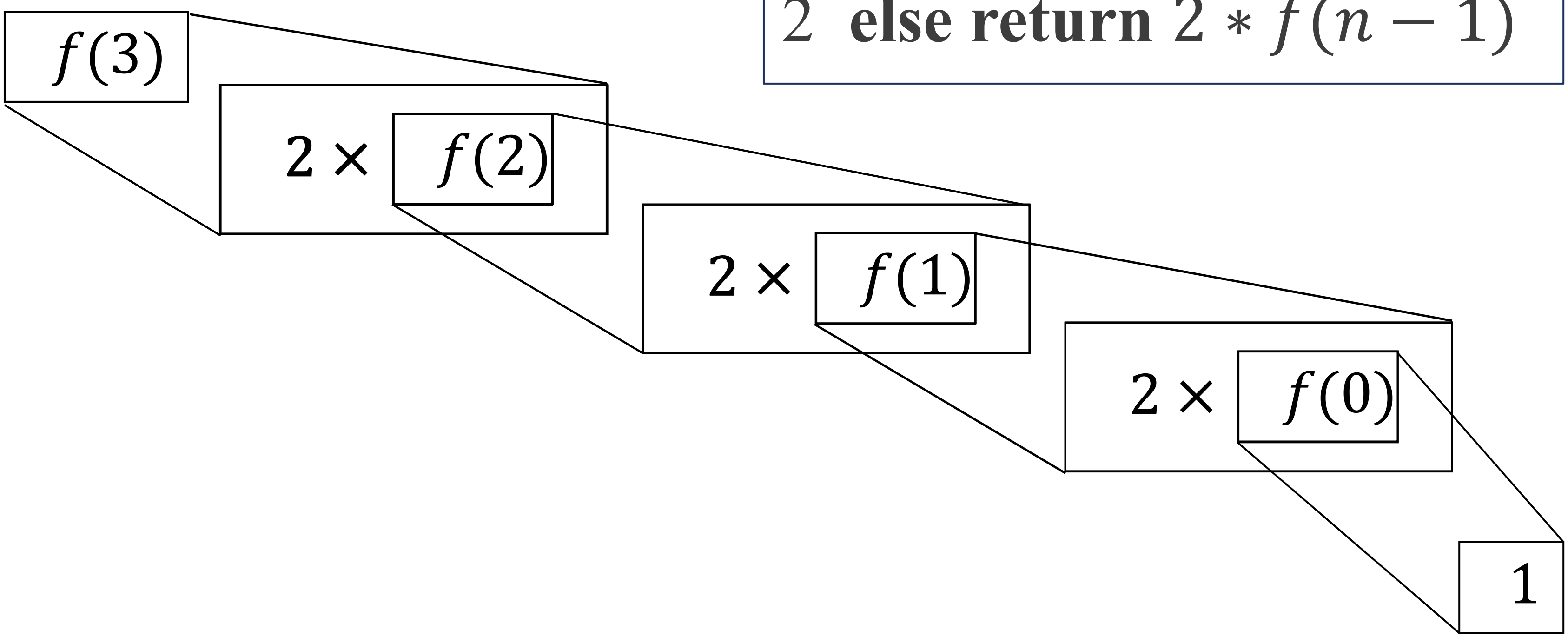
Recursion

Example 1 (cont'd)

$f(n)$

if $n = 0$ **then return** 1

else return $2 * f(n-1)$



Example 1 (cont'd)

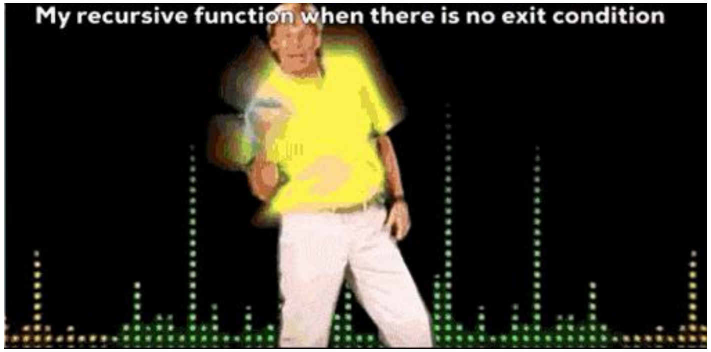
Correctness proof: 正确性证明

$f(k) = 2^k$循环不变量

- **Base case**: 递归出口
  - By definition, $f(0) = 2^0 = 1$, and the recursive algorithm returns 1 when $n = 0$. Therefore, the base case holds.
- **Inductive step** : 归纳假设
  - Assume that the property is true for $n = k$, i.e. $f(k) = 2^k$. We have to show that the property is true for $n = k + 1$.
  - By recursive algorithm, $f(k+1)$ returns $2 \times f(k) = 2 \times 2^k = 2^{k+1}$. So, inductive proof is complete.

Recursion

- $f(n) = 2 * f(n-1)$ is recursive definition of a function, which is defined in terms of itself. 递归函数的定义：根据自身性质来定义
- Therefore, to stop, there **must be a case** when it does not call itself (called base case 递归出口, stopping condition 终止条件 or exit condition) .出口很重要
- Recursion is an alternative to looping. As with looping, recursion can cause your program to loop forever. 递归是循环的一种替代方法。与循环一样，递归会导致程序永远循环。



Exit condition is very important for recursion…

Image source: https://gfycat.com/soulfulfairarthropods

Rules of Recursion 递归的规则

- Base cases 递归出口: Always have the base case (stopping condition 终止情况), which is solved without recursion.
  - Base case is usually the simplest case to solve.
- Making progress 每次递归都要离递归出口更近: for recursive cases, each new call must always make progress towards base case.
  - Sometimes you have the base case but it can never be reached.
- Design Rule: assume all recursive calls work.

  假设所有递归调用都有效

Efficiency of Recursion 递归的效率

递归算法对应的代码通常在逻辑上都非常简洁，因为我们只要定义最顶层的逻辑

- The nature of recursion is iteration. Therefore, any recursive function can be converted to an equivalent iterative (looping) method.

本质上递归是迭代，可以被转换成for循环方式

- Although recursion is elegant, it can be inefficient, because there are more calls to methods. 递归方法更加优雅写，但是开销更高
  - Sometimes, there are many recursive calls to the same instance. 可能关于同一个输入多次重复调用
- Iterative methods are more efficient and faster. 迭代方法效率更高

$f(n)$

$\quad total \leftarrow 1$

$\quad$ **for** $i \leftarrow 0$ **to** n **do**

$\quad\quad total \leftarrow total * 2$

$\quad$ **return** $total$

Iterative way to write $f(n)$

走台阶问题

- 思明校区建南大礼堂前有很多台阶，假设你每次登一级或者两级台阶，登到20级台阶有多少种走法?

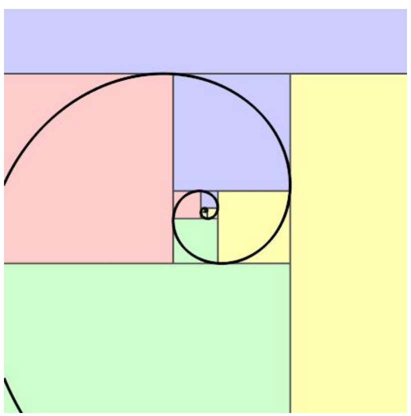
Recursion 刚才的走台阶问题

Example 2: Fibonacci sequence (斐波那契数列，又称黄金分割数列)

- Fibonacci sequence is defined by

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_n &= f_{n-1} + f_{n-2}, \quad \text{for } n \geq 2
\end{aligned}
$$

- 0, 1, 1, 2, 3, 5, 8, 13, 21….



Fib(n)
    if $n \leq 1$ **then**
        return n
    else
        return Fib($n-1$) + Fib($n-2$)

因为最后一步可走一步或两步，所以到达第n级台阶的方法数**f(n)**等于到达第n-1级台阶的方法数加上到达第n-2级台阶的方法数。这与斐波那契数列的定义一致，即序列中每个数都是前两个数的和。

Image source: https://en.wikipedia.org/wiki/Golden_spiral

复杂度分析

Example 2: Fibonacci sequence (cont'd)

- The recursive equation (递归方程) for the number of moves that solve the nth Fibonacci term is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + \mathbf{1} & \text{if } n > 1 \end{cases}.$$

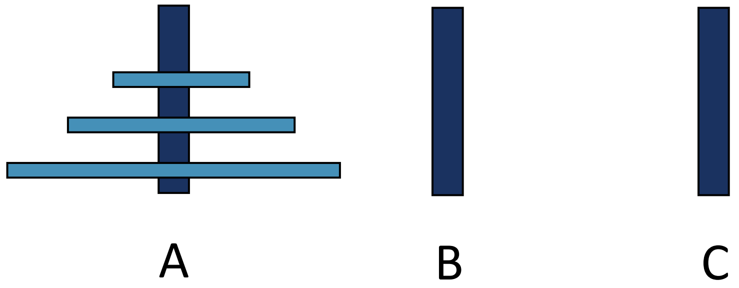
- Is it efficient to calculate the nth Fibonacci term by recursion?
  - When calculating Fib(5), how many times of Fib(3) and Fib(2) is calculated?

  在递归的时候可能会出现对同样输入用例的重复调用，但是我们在具体算法实现的时候，可以用数据结构来记录对于同一个输入用例的输出,以避免重复计算（动态规划）

  - https://visualgo.net/en/recursion?slide=1

Example 3: Towers of Hanoi (汉诺塔)

- **Objective:** Transfer disks from pole A to pole C. 有三根银色的柱子，目的就是将串在A柱上的金盘移动到C柱上
- **Rules:** Only move one disk at a time, and can’t put a bigger disk on a smaller one. 一次只能移动一个盘，并且大的盘子不能放到小盘子上



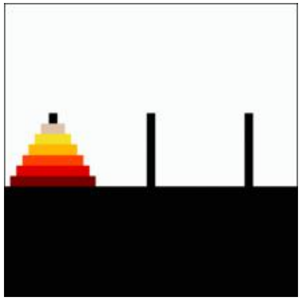


Image source: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Recursion

Example 3: Towers of Hanoi (cont'd)

- The recursive function $\mathrm{Hanoi}(n, A, B, C)$ means moving n disks from pole A to pole C using B as auxiliary.
- Steps: （步骤）
  - Move $n-1$ disks from A to B, using C as auxiliary.
  - Move the disk left on A directly to C.
  - Move the $n-1$ disks from B to C, using A as auxiliary.

```
Hanoi(n, A, B, C)
    if n = 1 then move(A, C)
    else
        Hanoi(n − 1, A, C, B)
        move(A, C)
        Hanoi(n − 1, B, A, C)
```

使用递归调用来解析

Illustration of recursion calls for $n = 3$

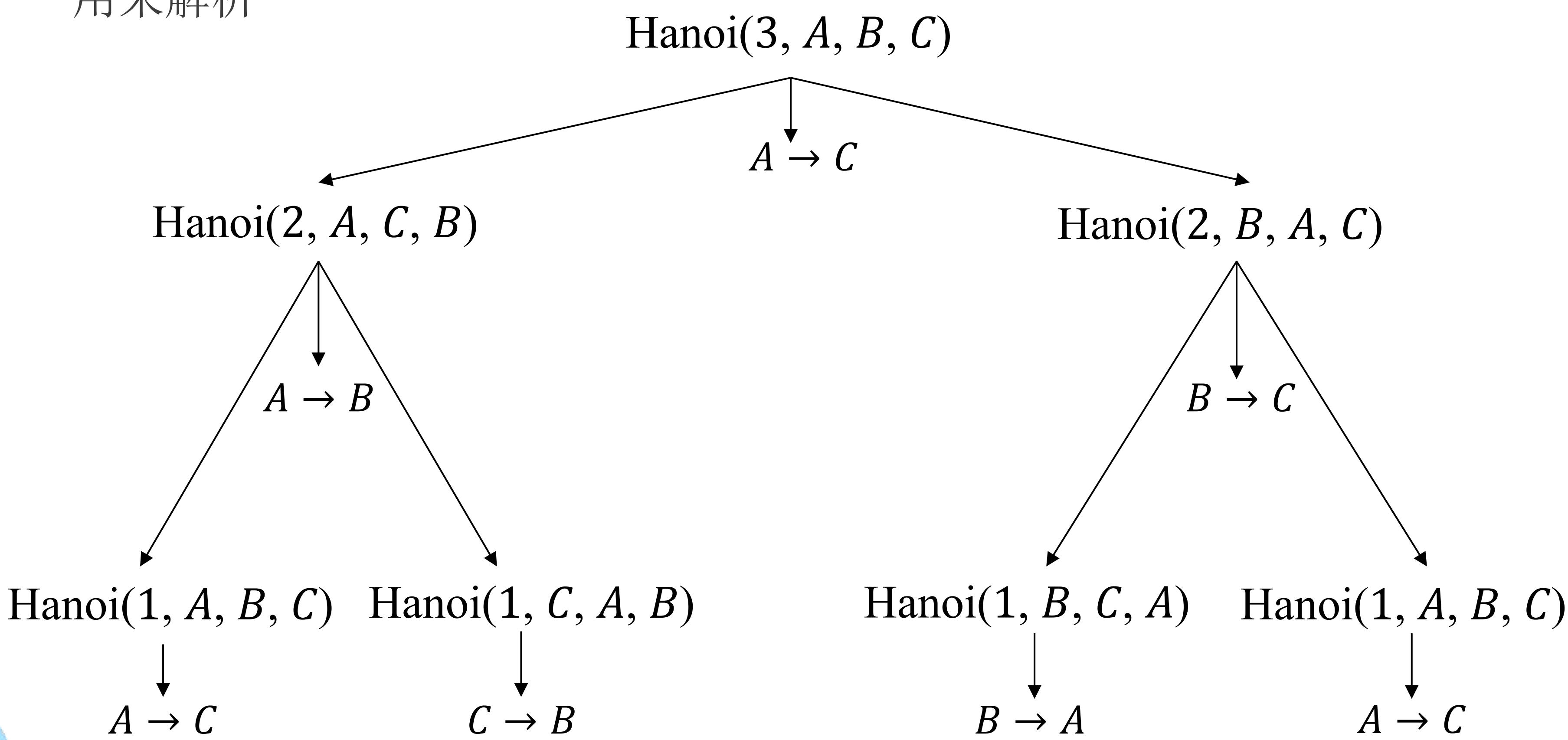


Illustration of recursion instances for $n = 4$

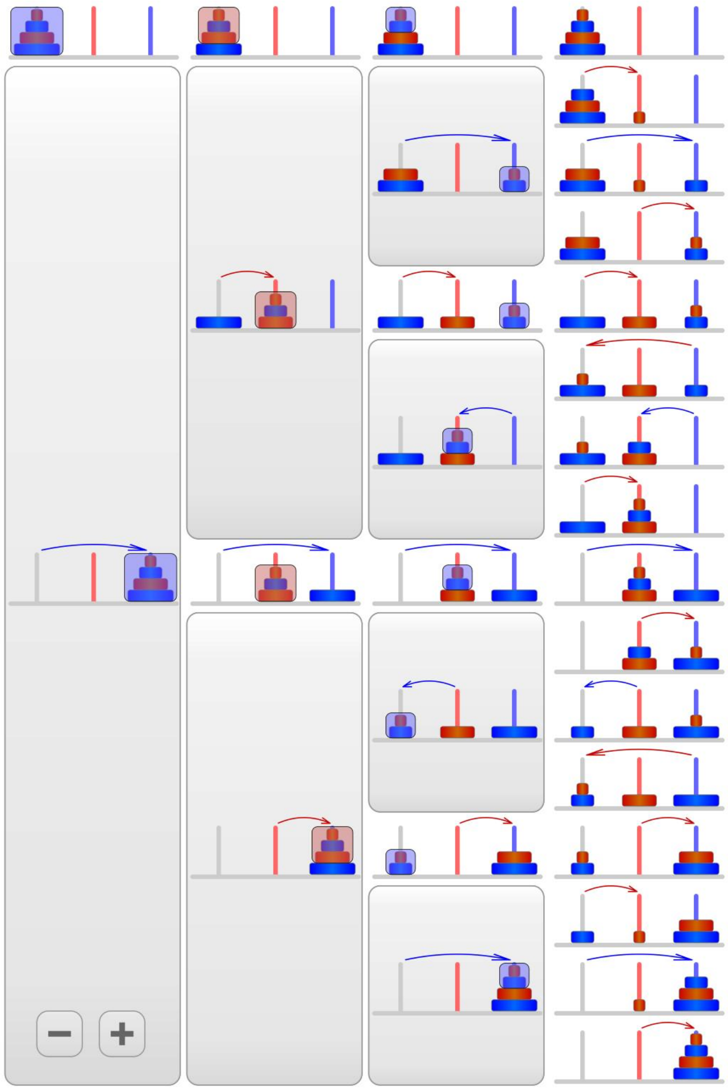


Image source: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Example 3: Towers of Hanoi (cont'd)

- **The recursive equation** for the number of moves that solve Towers of Hanoi is:

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

- However, it is a recursion equation, rather than a function of n. How to convert it as a function of n?(递归方程求解)

递归与数据结构

- 在编写递归算法的时候，我们需要把很多自顶向下过程中的中间状态一一保存，到了递归出口（base case）时，有要逐一回溯，直到最顶部
- 数据结构里面最适合用来存储这种中间状态的结构是?
  - 栈 （先进后出）

堆栈能够有效的记录中间一步步分解额复杂过程，
并且最后合并的过程和一开始拆解的过程正好相反。

Recursion 选择排序

Example 4: Selection sort (选择排序)

Similar to insertion sort, selecion sort is a very straightforward sorting algrotihm.

每次迭代都从当前位置往后，找到当前位置往后最小的元素，放到当前位置

步骤:

- Start with an empty left hand and the cards face down on the table.
- Then remove the smallest card at a time from the table, and insert it into the rightmost in the left hand.
- At all times, the cards held in the left hand are sorted.

选择排序对排列规模比较小的元素序列非常有效

选择排序算法 循环实现

SelectionSort(A)

for $i \leftarrow 1$ **to** $n-1$ **do**
    $k \leftarrow i$
    for $j \leftarrow i+1$ **to** n **do**
        if $A[j] < A[k]$ **then**
            $k \leftarrow j$
    if $k \neq i$ **then** $A[i] \leftrightarrow A[k]$

变量k用来存放最小元素的下标

i				*k*	
5	2	4	6	1	3

	i, k				
1	2	4	6	5	3

		i			*k*
1	2	4	6	5	3

			i		*k*
1	2	3	6	5	4

				i, k	
1	2	3	4	5	6

Example 4: Selection sort (cont'd)

- The recursive version of selection sort is **very easy to convert.**
- Replace the outer loop by a recursive call.**(用递归调用代替循环)**
  - Because we are actually doing the same thing for each subsequence $A[i \ldots n]$.
- 这里就是把原来的外部循环换成这种递归的方式
- Although it works, it is not elegant at all as a recursive algorithm.

递归算法实现选择排序

Usually, we **only write the changing variables** as the arguments of a recursive function in pseudocode.

RecursiveSelectionSort(i)

if $i \geq n$ **then return** 0 **出口**

else

    $k \leftarrow i$

    for $j \leftarrow i + 1$ **to** n **do**

        if $A[j] < A[k]$ **then**

            $k \leftarrow j$

    if $k \neq i$ **then** $A[i] \leftrightarrow A[k]$

    RecursiveSelectionSort($i + 1$)

Example 4: Selection sort 选择排序时间复杂度

- Selecting the minimal one among n elements needs $n-1$ comparisons.（n个元素中选择最小的需要N-1次循环）

- Therefore, the recursive equation（递归方程） is:

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

Example 5: Generating permutations 生成排列

Goal: Generate all $n!$ permutations of sequence $\langle 1,2,\dots,n\rangle$.

- What is a proper small instance of this problem?
  - Get all permutation of a sequence with $n-1$ elements. 如何划分小规模
- Given the solution of a small instance, how to solve the original problem?
  - Get all permutation of the sequence with n elements by the ones with $n-1$ elements. 小规模的解决，大问题的解决

策略1　固定位置放元素

Example 5: Generating permutations 生成排列

Idea 1: Put different elements on fixed position. 将不同的数放在固定位置

- Suppose we can generate all permutations for $n-1$ numbers.（生成n-1元素的所有排列）
- Generate all the permutations of the numbers $2,3,\ldots,n$ and add the number 1 to the beginning of each permutation (the ones starting with 1).
- Next, generate all permutations of the numbers $1,3,\ldots,n$ and add the number 2 to the beginning of each permutation (the ones starting with 2).
- Repeat this procedure until finally the permutations of $1,2,3,\ldots,n-1$ are generated and the number n is added at the beginning of each permutation.

Example 5: Generating permutations (cont’d)

Perm1(m)　　这里的**m**是指到了第几个**位置**

if $m = n$ **then** output $P[1..n]$

else

　　for $j \leftarrow m$ **to** n **do**　元素更新位置

　　　　$P[j] \leftrightarrow P[m]$　与m位置元素交换

　　　　Perm1($m+1$)　小规模问题(m+1->n)

　　　　$P[j] \leftrightarrow P[m]$　复位

Must switch back. Otherwise it will be messed up!

GeneratingPerm1()

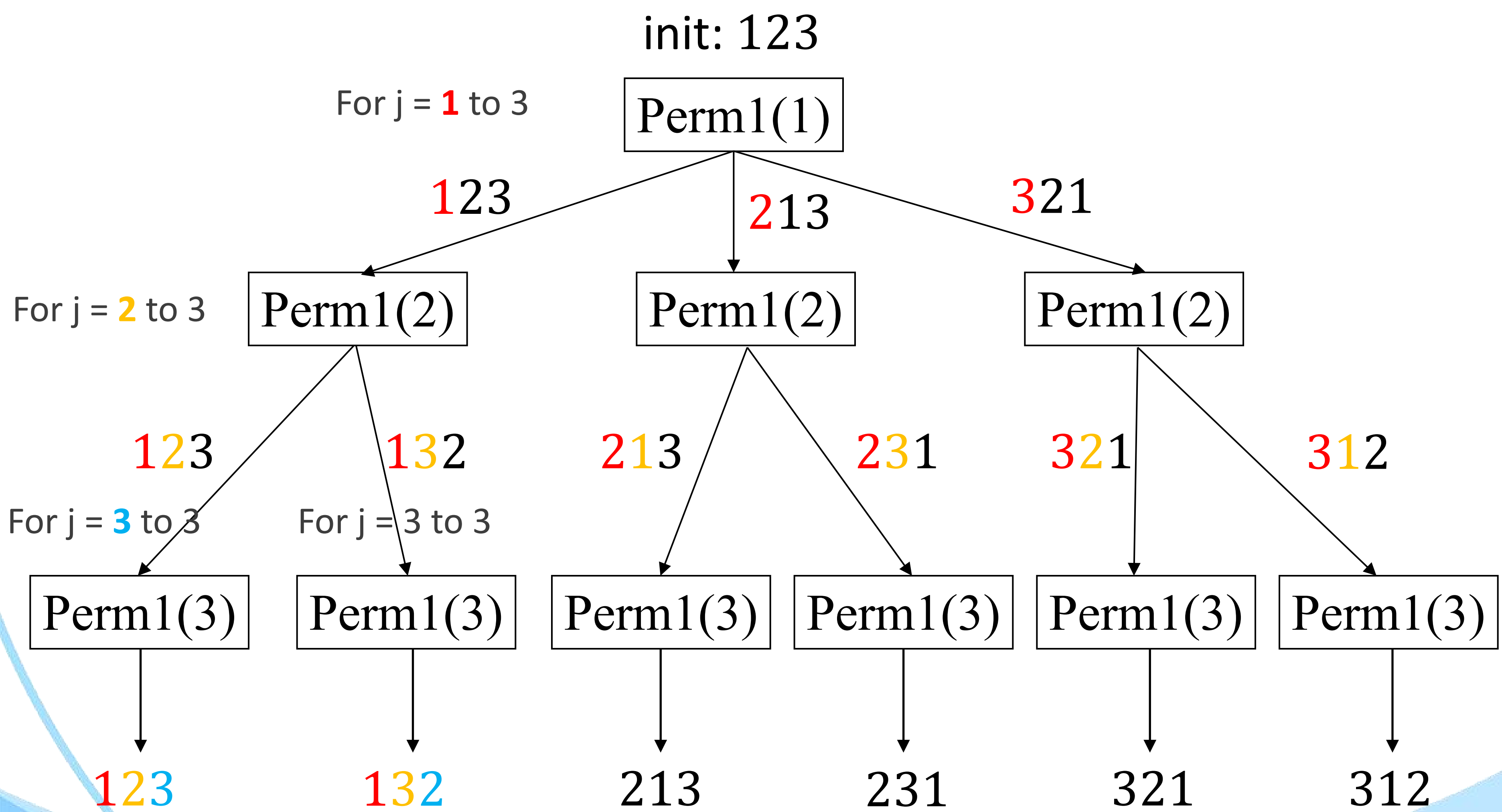
for $j \leftarrow 1$ **to** n **do**

　　$P[j] \leftarrow j$

Perm1(1)

从第1个位置开始

Illustration of recursion calls for $n = 3$



Try $n = 4$ by yourself

策略2 固定元素找位置

Example 4: Generating permutations (cont'd)

Idea 2: Put fixed element on different positions. 将固定的元素放到不同的位置

- Suppose we can generate all permutations of the numbers $1,2,\ldots,n-1$.
- First, we put n in $P[1]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[2\ldots n]$.
- Next, we put n in $P[2]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[1]$ and $P[3\ldots n]$.
- Then, we put n in $P[3]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[1\ldots 2]$ and $P[4\ldots n]$.
- Repeat the above process until finally we put n in $P[n]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[1\ldots n-1]$.

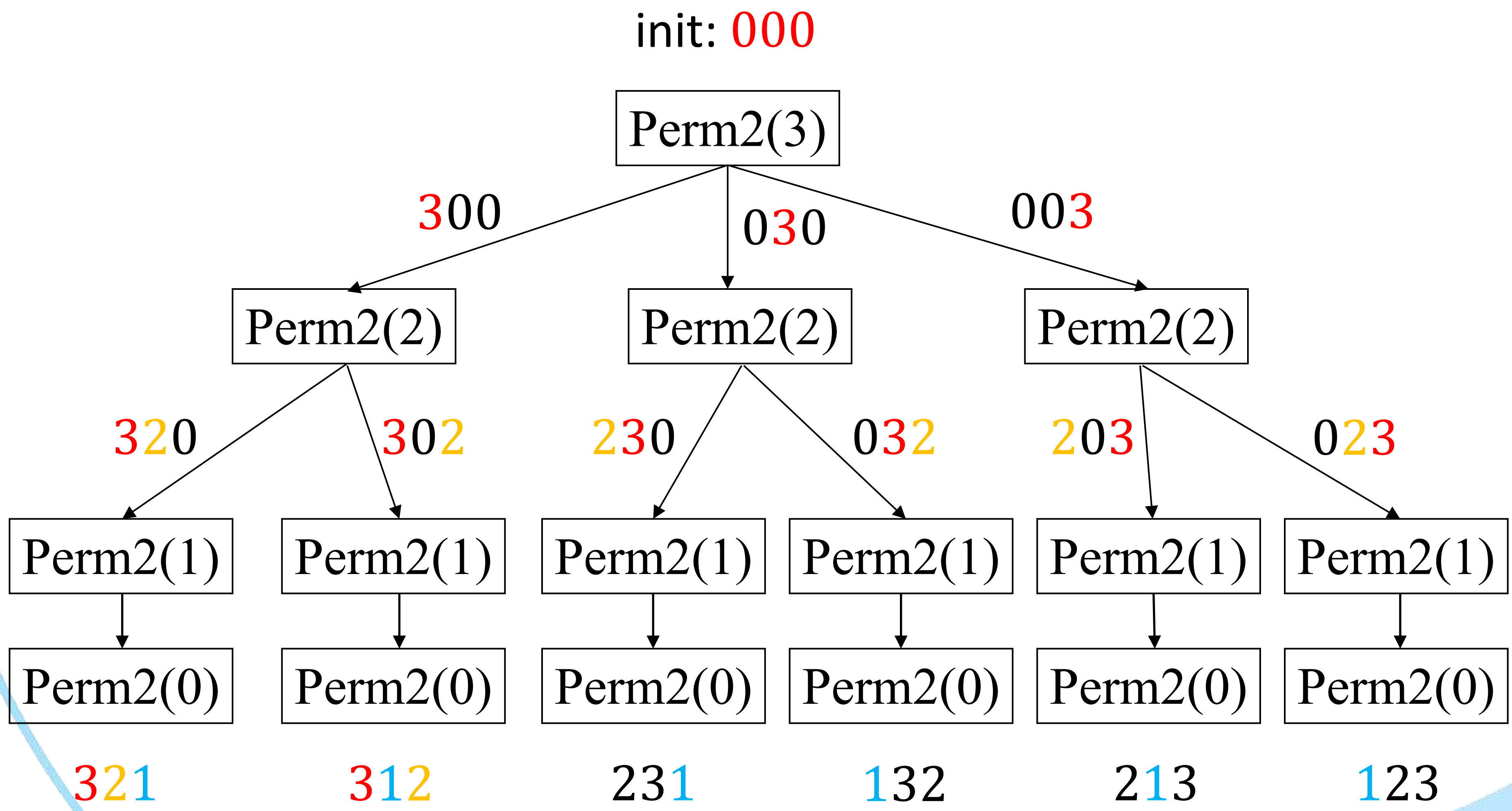
Example 5: Generating permutations (cont’d)

```
Perm2(m)            这里的m是元素（可放位置数目）从n个元素开始一直递减到0
if m = 0 then output P[1..n]
else                递归出口：要放0时，就输出
    for j ← 1 to n do    第j个位置(遍历)
        if P[j] = 0 then
            P[j] ← m     m元素放j位置
            Perm2(m − 1) 小规模（m-1->0)
            P[j] ← 0     复位
```

```
GeneratingPerm2()
for j ← 1 to n do
    P[j] ← 0      注意这里的初始化为0，而不是j
Perm1(n)
```

Must reset to 0. Otherwise the positions are not enough.

Illustration of recursion calls for $n = 3$



Try $n = 4$ by yourself

Example 5: Generating permutations (cont'd)

- For both ideas, each instance is split into n smaller instance with size $n-1$.
- Therefore, the recursive equation is:

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

Classroom Exercise （课堂练习）

Write the pseudocode of recursive linear search.

写线性搜索的递归版本伪代码

线性搜索是指针对某个数x，在一个数组里面从左到右搜索，如果找到就输出坐标的位置，如果没有就输出0

Classroom Exercise （课堂练习）

Solution:

RecursiveLinearSearch(i) i代表位置下标

if $i > n$ **then return** 0

if $A[i] = x$ **then**

  return i

else

  return RecursiveLinearSearch($i + 1$)

RECURSIVE ANALYSIS 递归分析

Recursive Analysis （递归分析）

递归分析的目的就是找到递归算法的时间复杂度

- Goal of recursion analysis: obtain an asymptotic bound Θ or O from the the recursive equation of a recursive algorithm.

$$T(n) = g\big(T(n-k)\big) \quad \text{or} \quad T(n) = g\big(T(n/k)\big)$$
$$\downarrow$$
$$T(n) = f(n)$$

Overview of Recursive Analysis Methods

基本思想：猜测递归方程的解，代入方程看是否存在满足的条件

- Substitution method (替换方法)
  - Guess a bound (directly guess or based on recursion tree); 基于猜测或者递归树方法判断出复杂度函数的上界
  - Prove our guess correct using Mathematical Induction.通过数学归纳法证明这个上界的正确性
- Master method (公式法)
  - **A theorem** with three cases; 有三种不同的情况
  - In each case, the result can be **directly obtained** without calculation.只要递归方程符合三种情况中一种，可以直接判断其复杂度函数

Technicalities 技术细节

In practice, we neglect certain technical details when we state and solve recursion. It won't affect the final asymptotic results.

对于一些技术细节，我们在渐进分析时忽略

- Suppose n is an non-negative integer in $T(n)$.
- Omit floors and ceiling. 取整操作，可以忽略顶和底
  - E.g. $T(n) = 2T(\lceil n/2 \rceil)$, and $T(n) = 2T(\lfloor n/2 \rfloor)$ are equivalent to $T(n) = 2T(n/2)$.
- As n is sufficiently small, we regard $T(n) = T(1)$, where $T(1)$ denotes the constant. 当n很小的时候，可以认为T(n)=T(1)
  - We can simply set $T(1) = 1$ and $T(0) = 0$.

Substitution Method 替换法

Steps of substitution method:

1. Guess the form of the solution. （猜复杂度的上界）
2. Use mathematical induction (数学归纳法) to find the constants and show that the solution works. （用数学归纳法来证明猜测是正确的）

Example 6

Consider the recursive equation for the number of comparisons of recursive selection sort: 用替换法来分析选择排序的递归版本复杂度

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

1. Guess $T(n) = O(n^2)$.
2. Prove: $T(n) \le cn^2$:
   - Base case: When $n = 1$, $T(1) = 1 \le c1^2$, for choosing $c \ge 1$. 初始步
   - Inductive step: Suppose $T(n-1) \le c(n-1)^2$. 归纳步

$$\begin{aligned} T(n) &\le c(n-1)^2 + n - 1 \\ &= cn^2 - 2cn + c + n - 1 \\ &\le cn^2 - 2cn + 2c + n - 1 \\ &= cn^2 - (2c-1)(n-1) \\ &\le cn^2 \text{ (for } c \ge \frac{1}{2}) \end{aligned}$$

Example 7

Consider the recursive equation for the number of moves that solve Towers of Hanoi: 替换法分析汉诺塔问题

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

1. Guess $T(n) = O(2^n)$.
2. Prove: $T(n) \le c2^n$:

- Base case: When $n = 1$, $T(1) = 1 \le c2^1$, for choosing $c \ge \frac{1}{2}$.
- Induction step: Suppose $T(n-1) \le c2^{n-1}$.

$$\begin{aligned} T(n) &\le 2c2^{n-1} + 1 \\ &= c2^n + 1 \\ &\cancel{\le} c2^n. \end{aligned}$$

- $T(n) \le c2^n + 1$ can't imply $\cancel{T(n)} \le c2^n$. How can we do?

(loose) (tight)

Substitution Method

- Sometimes the guess is correct, but somehow the math doesn‘t seem to work out in the induction.替换法的问题是有时猜测是正确的，但是数学归纳法推导不出来
- Usually, the problem is that the inductive assumption isn‘t strong enough to prove the detailed bound. 一般是因为假设的算法复杂度的上界O不够紧导致的
- Revise the guess by subtracting a lower-order term often permits the math to go through. 将猜测的上界减去一个低阶的项，这样能够用数学归纳法推导出算法的正确上界

Example 7 (cont'd)

- Consider the recursive equation for the number of moves that solve Towers of Hanoi: 继续考虑汉诺塔问题

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

1. Guess $T(n) = O(2^n)$.
2. Prove: $T(n) \le c2^n - b$: 这里将原来的上界减去了低阶项b
   - Base case: When $n = 1$, $T(1) = 1 \le c2^1 - b$, for choosing $c \ge \frac{1+b}{2}$.
   - Induction step: Suppose $T(n-1) \le c2^{n-1} - b$.

$$\begin{aligned} T(n) &\le 2(c2^{n-1} - b) + 1 \\ &= c2^n - 2b + 1 \\ &\le c2^n - b \text{ (for } b \ge 1). \end{aligned}$$

   - $T(n) \le c2^n - b$ can derive $T(n) \le c2^n$. Therefore $T(n) = O(2^n)$ is proved.

(tight) (loose)

替换法课堂练习

Use substitution method to give the asymptotic bound of the following recursive equation:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

Classroom Exercise

Solution:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

1. Guess $T(n) = O(n)$
2. Prove: $T(n) \le cn - b$:
   - Base case: When $n = 1$, $T(1) = 1 \le c - b$, for choosing any $c \ge 1 + b$.
   - Inductive step: Suppose $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor - b$ and $T(\lceil n/2 \rceil) \le c\lceil n/2 \rceil - b$.

$$\begin{aligned} T(n) &\le c\lfloor n/2 \rfloor - b + c\lceil n/2 \rceil - b + 1 \\ &= cn - 2b + 1 \\ &\le cn - b \text{ (for } b \ge 1\text{)} \end{aligned}$$

   - $T(n) \le cn - b$ can derive $T(n) \le cn$. Therefore $T(n) = O(n)$ is proved.

Example 8

$$T(n) = 8T(n/2) + 5n^2$$

1. Guess $T(n) = O(n^3)$.
2. Prove: $T(n) \leq cn^3$:
   - Base case: When $n = 1$, $T(1) = 1 \leq c$, for choosing any $c \geq 1$.
   - Inductive step: Suppose $T(n/2) \leq c(n/2)^3$.

$$\begin{aligned} T(n) &\leq 8c(n/2)^3 + 5n^2 \\ &= cn^3 + 5n^2 \end{aligned}$$

   - $T(n) \leq cn^3 + 5n^2$ can't prove $T(n) \leq cn^3$. We should subtract a lower-order term.

Example 8 (cont'd)

$$T(n) = 8T(n/2) + 5n^2$$

1. Guess $T(n) = O(n^3)$.
2. Prove: $T(n) \leq cn^3 - bn^2$:

- Base case: When $n = 1$, $T(1) = 1 \leq c - b$, for choosing any $c \geq 1 + b$.
- Inductive step: Suppose $T(n/2) \leq c(n/2)^3 - b(n/2)^2$.

$$\begin{aligned} T(n) &\leq 8[c(n/2)^3 - b(n/2)^2] + 5n^2 \\ &= cn^3 - 2bn^2 + 5n^2 \\ &= cn^3 - bn^2 - bn^2 + 5n^2 \\ &\leq cn^3 - bn^2 \text{ (for } b \geq 5) \end{aligned}$$

- $T(n) \leq cn^3 - bn^2$ can derive $T(n) \leq cn^3$. Therefore $T(n) = O(n^3)$ is proved.

Example 9

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

1. Guess $T(n) = O(n)$.
2. Prove: $T(n) \le cn$:

- Base case: When $n = 1$, $T(1) = 1 \le c1$, for choosing $c \ge 1$.
- Inductive step: Suppose $T(n/2) \le c(n/2)$.

$$T(n) \le cn + n$$
$$= O(n)?$$

这里只是证明了T(n)<=(c+1) n， 不是T(n) <= c n

- Wrong! The error is that we haven‘t proved the exact form of the inductive hypothesis, i.e. $T(n) \le cn$.
- 试下将算法复杂度上界猜测为更加低阶或者高阶的

Example 9 (cont'd)

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

1. Guess $T(n) = O(n \lg n)$.
2. Prove: $T(n) \leq cn \lg n$:
   - Base case: When $\boldsymbol{n = 2},$ $T(2) = 2T(1) + 2 = 4 \leq c2 \lg 2$, for choosing $c = 2$.
   - Inductive step: Suppose $T(\lfloor n/2 \rfloor) \leq c(\lfloor n/2 \rfloor) \lg (\lfloor n/2 \rfloor)$. **初始项不一定都选n=1**

$$\begin{aligned} T(n) &\leq 2c(\lfloor n/2 \rfloor) \lg (\lfloor n/2 \rfloor) + n \\ &\leq cn \lg (n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \ \ (\text{for } c \geq 1) \end{aligned}$$

Example 9 (cont'd)

- In the above proof, we set $n = 2$ at the base case.
- Actually, we usually don't need to set $n = 1$ for all base cases, because it sometimes doesn't work. 归纳法中的n不需要每次都选1，选其他小的数也可以
  - e.g. can't prove $T(1) = 1 \leq c1\lg 1 = 0$.
- The asymptotic analysis only requires us to prove for some $n \geq n_0$. Therefore, it is ok to set $n = 2$ or $n = 3$ at the base case.

Substitution Method: Changing Variables

变量代换

Sometimes, a little algebraic manipulation can make an unknown recursion similar to one you have seen before. 有的时候通过将递归公式里面的n换成成为其他变量的函数，可能会简化计算

Example 10

$$\boldsymbol{T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n}$$

- Renaming $m = \lg n$ yields $n = 2^m$ and:

$$\boldsymbol{T(2^m) = 2T(2^{m/2}) + m.}$$

- We can now rename $S(m) = T(2^m)$ to produce the new recursion:

$$\boldsymbol{S(m) = 2S(m/2) + m,}$$

由$(T(2^m) = S(m); T(2^{m/2}) = S(m/2))$

which has a solution of $S(m) = O(m\lg m)$. （前题结论例4.1）

- Changing back from $S(m)$ to $T(n)$, we obtain:

$$T(n) = T(2^m) = S(m) = O(m\lg m) = O(\lg n\lg\lg n).$$

由 $m = \lg n$

Substitution Method

How to make a good guess:

- Bad News:
  - No general way to guess the correct solutions to recursion. 没有通用的方法来产生一个正确的算法时间复杂度猜测
  - Good guess = E (experience) + C (creativity) + L (luck).
- Good News:
  - Recursion tree often generates good guesses. 递归树可以用于产生一个比较好的猜测

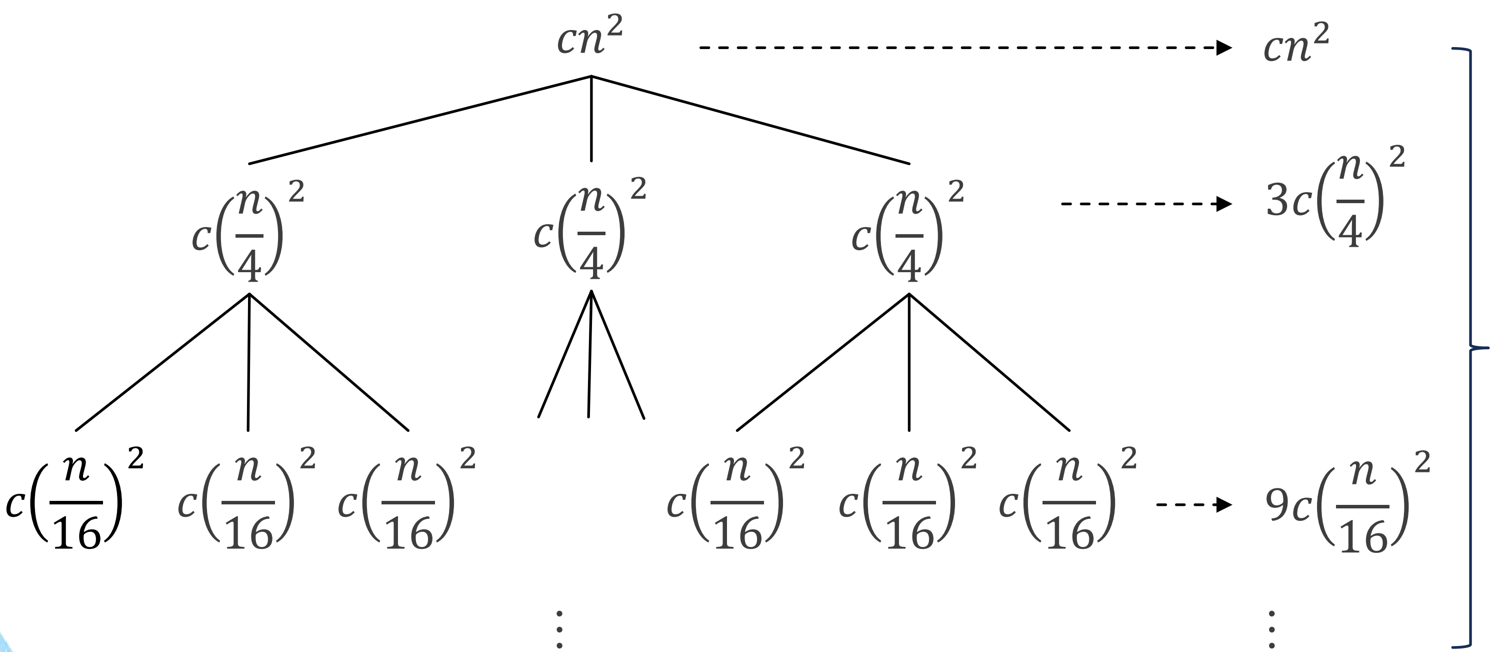
RECURSION TREE 递归树

Recursion Tree (递归树)

- The recursion-tree is a straightforward way to devise a good guess. 递归树是一个非常直观的用于分析递归算法复杂度的方法
- Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm. 当递归涉及到分治的时候，递归树非常有用
- In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- 将递归树上所有层上的所有节点的cost加起来就就可以得到一个估计
    1. We sum all the per-node costs within each level of the tree to obtain a set of *per-level costs*;
    2. We sum all the per-level costs to determine the total cost of all levels of the recursion.
- Notice: Recursion tree only provides a guess. It is not a strict proof. Substitution method is still needed after we guess a bound by recursion tree. （递归树这种方式不是一个严格的证明，我们还是要用之前的替代法的方式来证明算法的界）

Example 11

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$$



What is the height of the tree?

Recursion Tree

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$$

Example 11 (cont'd)

- The cost sequence of each level is: (每层代价)

$$cn^2, c(n/4)^2, c(n/4^2)^2, \ldots, c(n/4^i)^2$$

- Denote height of the recursion tree as k. (共k层)
- The node at the leaf of the tree is 1. Therefore the leaf is achieved when $(n/4^k) = 1$ and thus $k = \log_4 n$.

We can simply assume that n is an exact power of 4.

展开到第K层，其规模$n/4^k=1$,$k=\log_4 n$

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$$

Example 11 (cont'd)

- Summing up all levels, the total cost is:

等比数列
求和公式

$$T(n) = cn^2 + 3c\left(\frac{n}{4}\right)^2 + 9c\left(\frac{n}{16}\right)^2 + 27c\left(\frac{n}{64}\right)^2 + \dots$$

$$= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2$$

$$= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 = \frac{1}{1-3/16} cn^2 = O(n^2)$$

Formula of infinity geometric series (无穷几何级数)

Example 11 (cont'd)

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$$

- Notice again: Recursion tree only provides a guess. It is not a strict proof. We still need substitution method:

1. Guess $T(n) = O(n^2)$.
2. Prove: $T(n) \le dn^2$:

Why do we use d here rather than c?

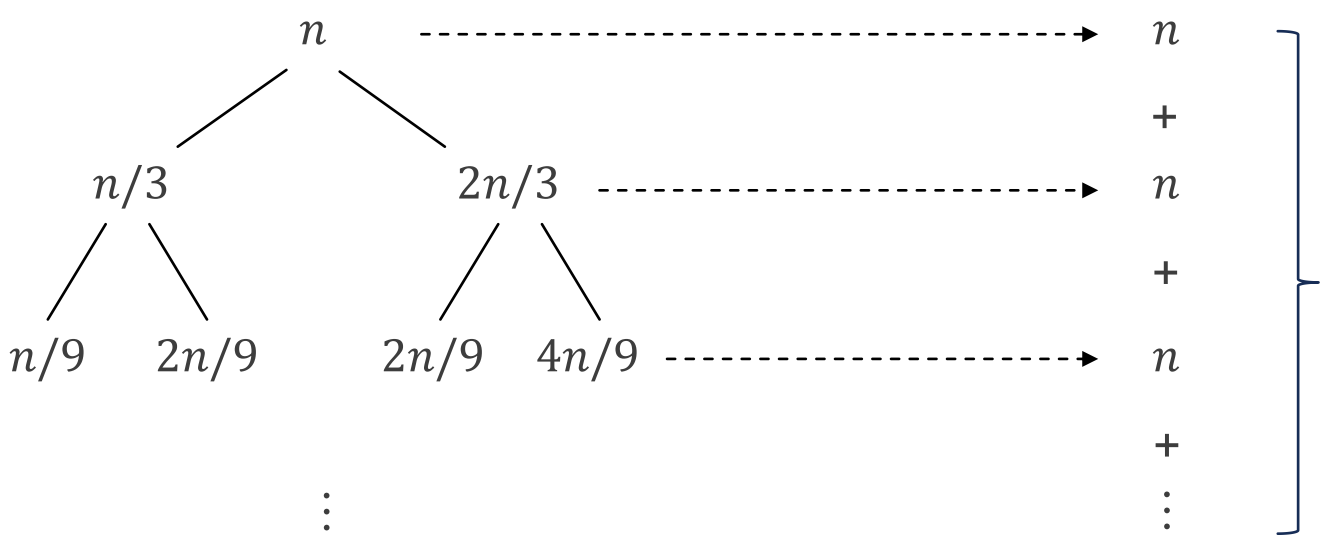
- Base case: When $n = 1$, $T(1) = 1 \le d1^2$, for choosing $d \ge 1$.
- Inductive step: Suppose $T(n/4) \le d(n/4)^2$.

$$\begin{aligned} T(n) &\le 3d\left(\frac{n}{4}\right)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 \text{ (for } d \ge \frac{16}{13}c) \end{aligned}$$

因为递归公式里面有c

Example 12

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



What is the height of the tree?

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

Example 12 (cont'd)

- If there are different decreasing rate, e.g. $n/3$ and $2n/3$ in this example, we should determine the slowest deceasing rate.
  - The one with slowest deceasing rate goes deepest.
- $2n/3$ is the slowest one. Therefore, the height is calculated by: 用树最长的分支来估计树的高度（高估）

$$\left(\frac{2}{3}\right)^k n = 1$$
$$k = \log_{3/2} n$$

- As observed from the tree, the cost of each level is n. But not all levels have cost n because some branches with faster decreasing rate may reach the leaves earlier. The total cost is:

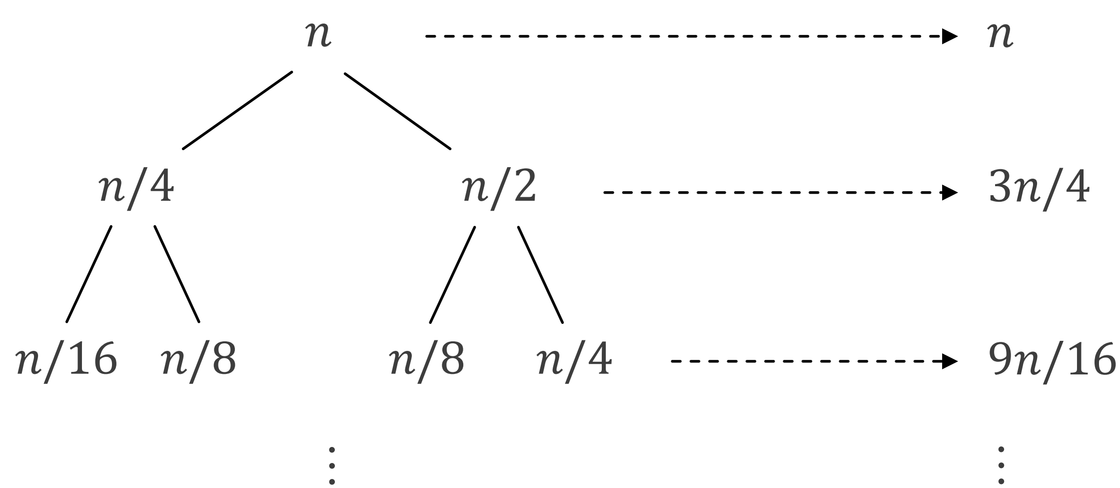
$$T(n) \le n(k+1) \le n(\log_{3/2} n + 1) = O(n \lg n).$$

Classroom Exercise

Use recursion tree to guess the asymptotic bound of the following recursion equation:

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

Solution:



- The slowest deceasing rate is $n/2$.
- The height is calculated by: $(1/2)^k n = 1$ and $k = \lg n$.

$$T(n) \le n + \frac{3}{4}n + \left(\frac{3}{4}\right)^2 n + \dots + \left(\frac{3}{4}\right)^{\lg n} n$$

$$< \frac{1}{1 - 3/4} n = 4n = O(n).$$

MASTER METHOD 公式法

Master Method 公式法

下列形式的递归方程：将一个规模为n的问题划分为a个规模为n/b的子问题

- The master method provides a "cookbook" method for solving recurrences of the form
$$T(n) = aT(n/b) + f(n).$$
  - $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. 渐进正函数
- The recursion form describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b. 解每个子问题所需时间为$T(n/b)$
- The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$.

Master Method

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recursion

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically with three cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\big(f(n)\big)$.

递归解是由两个函数中，数量级较大的一个决定

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

What does the master theorem mean?

- In each of the three cases, we are comparing $f(n)$ with $n^{\log_b a}$.
- Intuitively, the solution to the recursion is determined by the order of these two functions.
  - If, as in case 1, $n^{\log_b a}$ has high order , then the solution is $T(n) = \Theta(n^{\log_b a})$.
  - If, as in case 2, the two functions are the same order, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n)$.
  - If, as in case 3, $f(n)$ has high order, then the solution is $T(n) = \Theta\big(f(n)\big)$.

Master Method

In short:

- Comparing $f(n)$ with $n^{\log_b a}$, choose the larger order one with big Θ.
- If they have the same order, multiply with $\lg n$.

Master Method

Take a deeper look of the master theorem. Beyond this intuition of comparing order of functions, there are some technicalities that must be understood. （除了直觉理解，还有一些细节）

- In case 1, not only must $f(n)$ have lower order than $n^{\log_b a}$, its order must be polynomially lower. 多项式小 （小n^{ϵ} ）
  - The order of $f(n)$ must be asymptotically lower than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant 对于某个常数，f(n)渐进地比$n^{\log_b a}$小n^{ϵ} 倍
- In case 3, not only must $f(n)$ have higher order than $n^{\log_b a}$, its order must be polynomially higher多项式大 （大n^{ϵ} ），and in addition satisfy the "regularity" condition that $af(n/b) \le cf(n)$.
  - The order of $f(n)$ must be asymptotically higher than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$.
  - No worry about $af(n/b) \le cf(n)$, it holds for most of the cases.

Example 13

$$T(n) = 9T(n/3) + n$$

- We have $a = 9$, $b = 3$, $f(n) = n$, and thus we have $n^{\log_b a} = n^{\log_3 9} = n^2$.
- We thus compare n and n^2.
- Since $f(n) = n = O(n^{\log_3 9 - \epsilon})$ for $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^2)$.

Example 14

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

- We have $a = 1$, $b = 3/2$, $f(n) = 1$, and thus we have $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
- We thus compare 1 and 1.
- Since $f(n) = 1 = \Theta(1)$, we can apply case 2 and thus the solution to the recursion is $T(n) = \Theta(\lg n)$.

Master Method

Example 15

$$T(n) = 3T(n/4) + n\lg n$$

- We have $a = 3$, $b = 4$, $\boldsymbol{f(n) = n\lg n}$, and thus we have $\boldsymbol{n^{\log_b a} = n^{\log_4 3} \approx n^{0.793}}$.
- We thus compare $n\lg n$ and $n^{\log_4 3}$.
- Since $f(n) = n\lg n = \Omega(n) = \Omega(n^{\log_4 3+\epsilon})$ for $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$.
- For sufficiently large n,

  $$af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n\lg n = cf(n) \text{ for } c = 3/4.$$
- Consequently, by case 3, the solution to the recursion is $T(n) = \Theta(n\lg n)$.

$af(n/b) \le cf(n).$

$\log_4 3 \approx 0.8$

Master Method

- 公式法并不能覆盖所有的递归公式的情况
- The three cases do not cover all the possibilities for $T(n)$.
- There is a gap between cases 1 and 2 when the order of $f(n)$ is lower than $n^{\log_b a}$ but not polynomially lower. 在情况1之间和情况2之间有个gap，如果f(n)不是多项式的小于$n^{\log_b a}$
- Similarly, there is a gap between cases 2 and 3 when the order of $f(n)$ is higher than $n^{\log_b a}$ but not polynomially higher. 同样情况2和情况3之间也有一个gap
- If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recursion.

Master Method

- Master method is used for the following form of recursion equation 公式法能用于指导优化递归算法

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

- We compare $\boldsymbol{n^{\log_b a}}$ with $\boldsymbol{f(n)}$ and select **the larger one**.
- Therefore, to reduce the cost of a recursive algorithm, we can:
  - Reduce $f(n)$: reduce the cost of computation in each recursion call.
  - Reduce a: reduce the number of recursion calls.
  - Increase b: reduce the size of small instance.

Can we use master method to give the asymptotic bound of the following recursive equation?

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

Solution:

The master method does not apply to the recursion in the following example.

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

- Even though it has the proper form: $a = 2$, $b = 2$, $f(n) = n\lg n$, and $n^{\log_b a} = n$.
- We thus compare $n\lg n$ and n.
- It might seem that case 3 should apply, since the order of $f(n) = n\lg n$ is asymptotically higher than n. The problem is that it is not polynomially higher.
- We can't find a constant $\epsilon > 0$ such that $f(n) = n\lg n = \Omega(n^{1+\epsilon}) = \Omega(n \cdot n^{\epsilon})$

多项式求值

Example 16: Polynomial Evaluation

- Given a polynomial function

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1},$$

  We want to calculate the value of $p(x)$ at some point x_0.

- We can use Horner's rule (秦九韶算法, 霍纳法则) recursively evaluates the polynomial function by rewriting as:

$$p(x) = a_0 + x\big(a_1 + x(a_2 + \dots + x(a_{n-2} + x a_{n-1})\dots)\big).$$ （反过来 看）

  Let

$$A_i = \begin{cases} a_{n-1} & i = 1 \\ A_{i-1} x_0 + a_{n-i} & i > 1 \end{cases}$$

Empirical Experiment

Example 16 (cont'd)

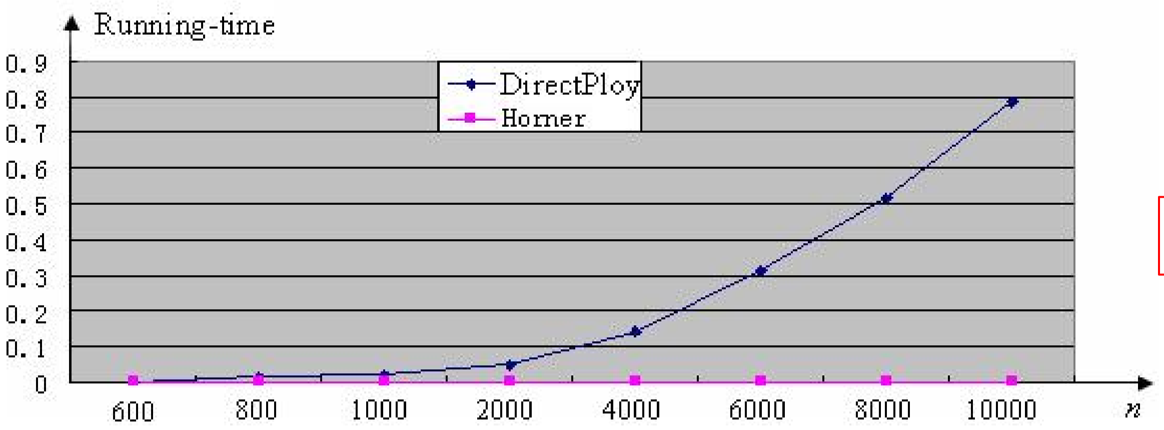
Horner(A, x_0, i)
if $i = 1$ **then return** a_{n-1}
else
    return $a_{n-i} + x_0 * \text{Horner}(A, x_0, i-1)$

DirectPoly(A, x_0)
$total \leftarrow a_0$
for $i \leftarrow 1$ **to** $n-1$ **do**
    $total \leftarrow total + a_i * \text{power}(x_0, i)$
return $total$

Example 16 (cont'd)

- Running-time comparison of DirectPloy and Horner:

n	600	800	1000	2000	4000	6000	8000	10000
DirectPloy	0.0	0.015	0.018	0.046	0.141	0.312	0.515	0.785
Horner	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0



Why?

Conclusion

After this lecture, you should know:

- How to devise a recursive algorithm?
- What is a recursive equation?
- How to derive the asymptotic result from the recursive equation?
- How to draw a recursive tree?

谢谢

有问题欢迎随时跟我讨论

Homework （课后作业）

- Page 48-49

  4.3

  4.5

  4.7

  4.12

  4.15