武汉大学
WUHAN UNIVERSITY

数据结构与算法

递归

张晓平

武汉大学数学与统计学院

Table of contents

递归的一些例子

递归的一些例子

阶乘函数

阶乘函数

定义：标准

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1) \cdots 2 \cdot 1, & n \geq 1. \end{cases}$$

定义：递归

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \geq 1. \end{cases}$$

阶乘函数

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

阶乘函数

在 Python 中，每一次函数被调用，就会创建一个活动记录 (activation record)，它包含了这个函数的所有与调用有关的数据，包括参数和本地变量.

阶乘函数

当函数的执行导致一次嵌套函数的调用时，将挂起前一个调用的执行，并且其活动记录将存储源码在控制流从嵌套调用返回时的位置．此过程既适用于一个函数调用另一个函数的标准情形，也适用于函数调用自身的递归情况。关键在于每次调用都有不同的活动记录。

阶乘函数

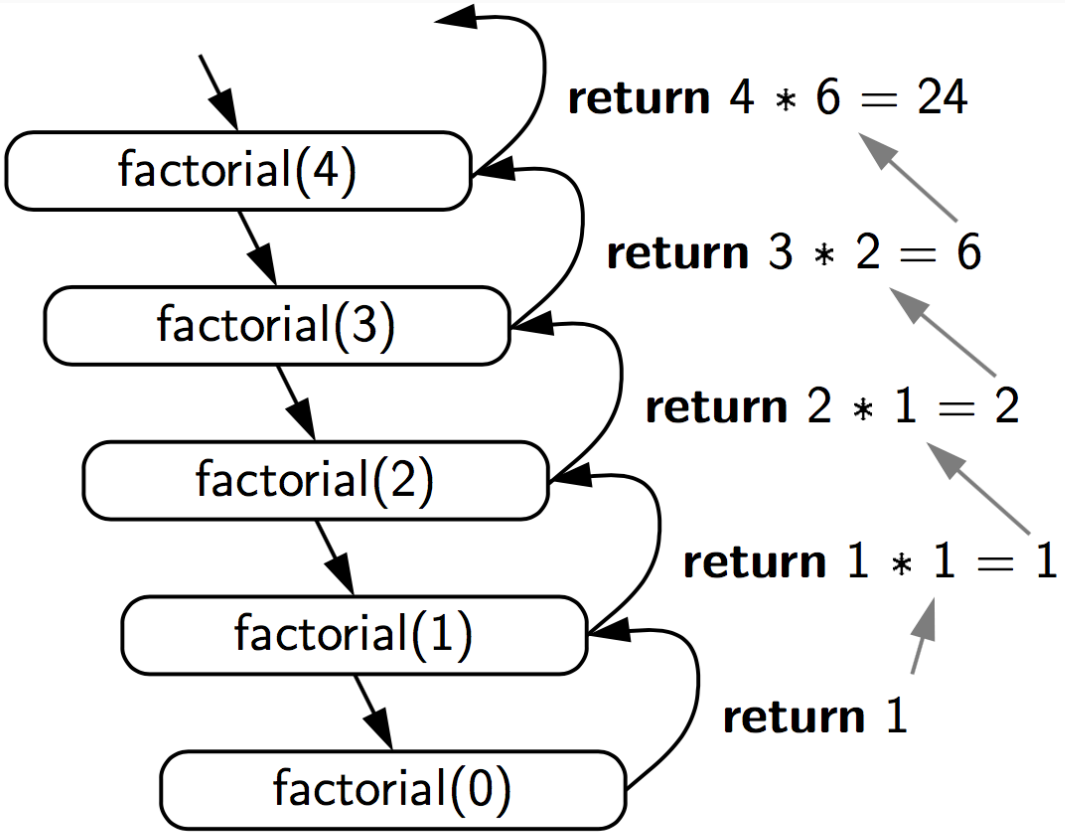

Figure 4.1: A recursion trace for the call factorial(5).

阶乘函数

算法分析

- 共计 $n+1$ 次调用;
- 每次调用都执行常数次的操作.

时间复杂度为 $O(n)$.

递归的一些例子

英国标尺

英国标尺

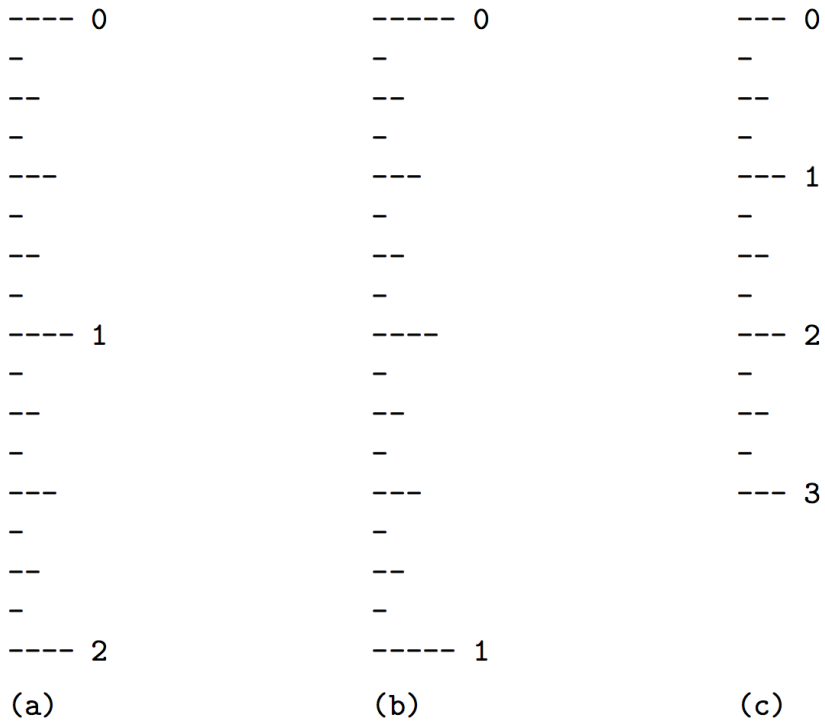

Figure 4.2: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

英国标尺

中心刻度长度为 $L \geq 1$ 的区间由以下几部分组成：

- 中心刻度长度为 $L-1$ 的区间
- 长度为 L 的刻度
- 中心刻度长度为 $L-1$ 的区间

英国标尺 I

```
def draw_line(tick_length, tick_label=''):
  line = '-' * tick_length
  if tick_label:
    line += ' ' + tick_label
  print(line)
```

英国标尺 I

```
def draw_interval(center_length):
  if center_length > 0:
    draw_interval(center_length - 1)
    draw_line(center_length)
    draw_interval(center_length - 1)
```

英国标尺 I

```
def draw_ruler(num_inches, major_length):
  draw_line(major_length, '0')
  for j in range(1, 1 + num_inches):
    draw_interval(major_length - 1)
    draw_line(major_length, str(j))
```

英国标尺

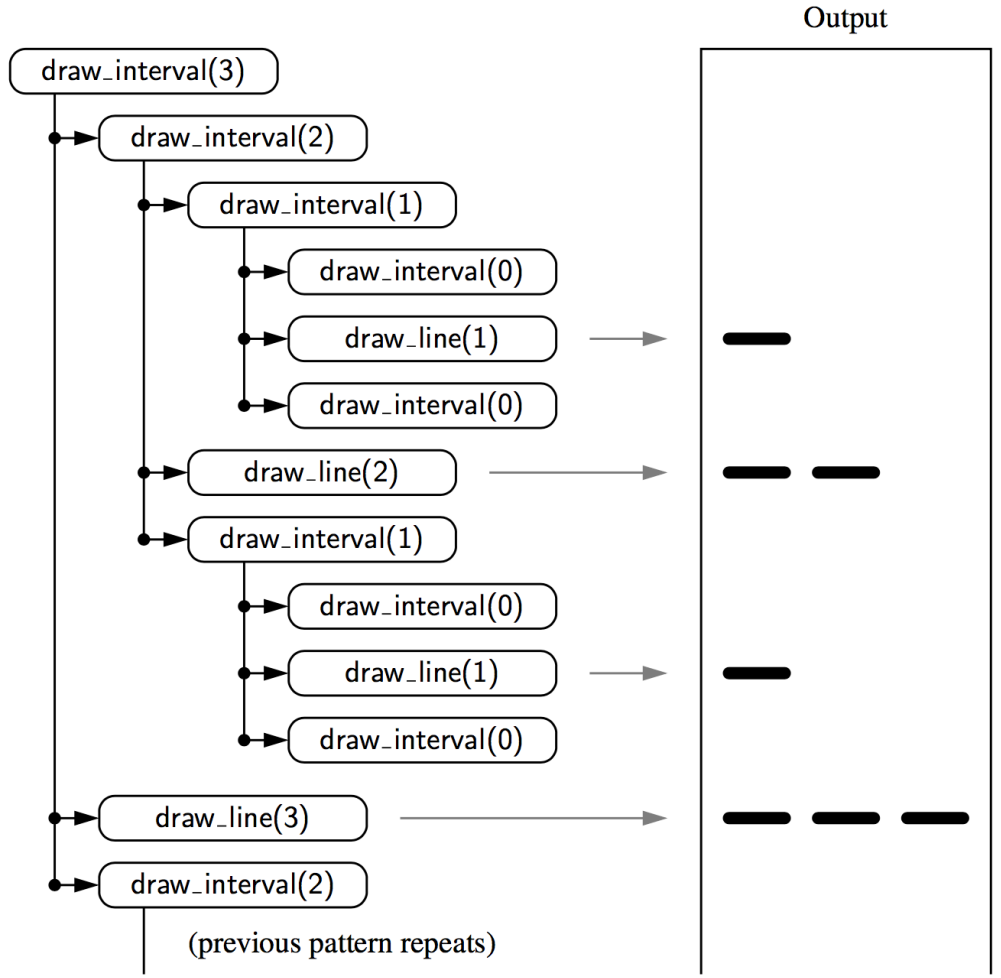

Figure 4.3: A partial recursion trace for the call draw_interval(3). The second pattern of calls for draw_interval(2) is not shown, but it is identical to the first.

英国标尺: Algorithm Analysis

问题

调用`draw_interval(c)`将生成多少条线条?

答

因调用一次 `draw_interval(c)` 将调用两次 `draw_interval(c-1)` 和一次 `draw_line`，故调用一次 `draw_interval(c)` 将生成 $2^c - 1$ 条线段.

递归的一些例子

折半查找

折半查找

折半查找用于在一个长度为 n 的已排序的序列中定位某目标值.

这是计算机算法中最重要的一种，也是我们经常按顺序存储数据的原因。

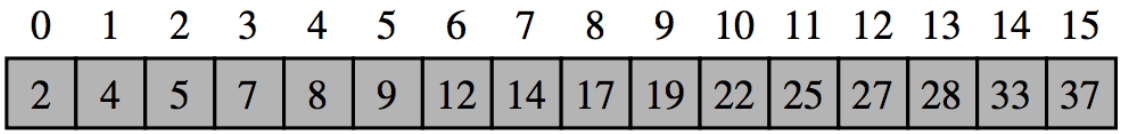

Figure 4.4: Values stored in sorted order within an indexable sequence, such as a Python list. The numbers at top are the indices.

折半查找

若序列未排序，可使用顺序查找法来定位某个目标值．顺序查找法依次遍历序列中的各个元素，直到目标值被找到或序列遍历完毕．该算法的时间复杂度为 $O(n)$.

折半查找

折半查找包含两个参数，即 low 和 $high$，使得候选元素的索引位于两者之间.

- 令 $low = 0$, $high = n - 1$.
- 比较目标值和中位值 $data[mid]$, $mid = \lfloor (low + high)/2 \rfloor$.
- 考虑以下三种情形：
  - 若 $target == data[mid]$，查找成功；
  - 若 $target < data[mid]$，继续在序列的左半部分查找；
  - 若 $target > data[mid]$，继续在序列的右半部分查找.

若出现 $low > high$，即区间 $[low, high]$ 为空，则查找失败.

折半查找 I

```
def binary_search(data, target, low, high):
  if low > high:
    return False
  else:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      return binary_search(data, target, low,
      mid - 1)
    else:
      return binary_search(data, target, mid +
      1, high)
```

折半查找

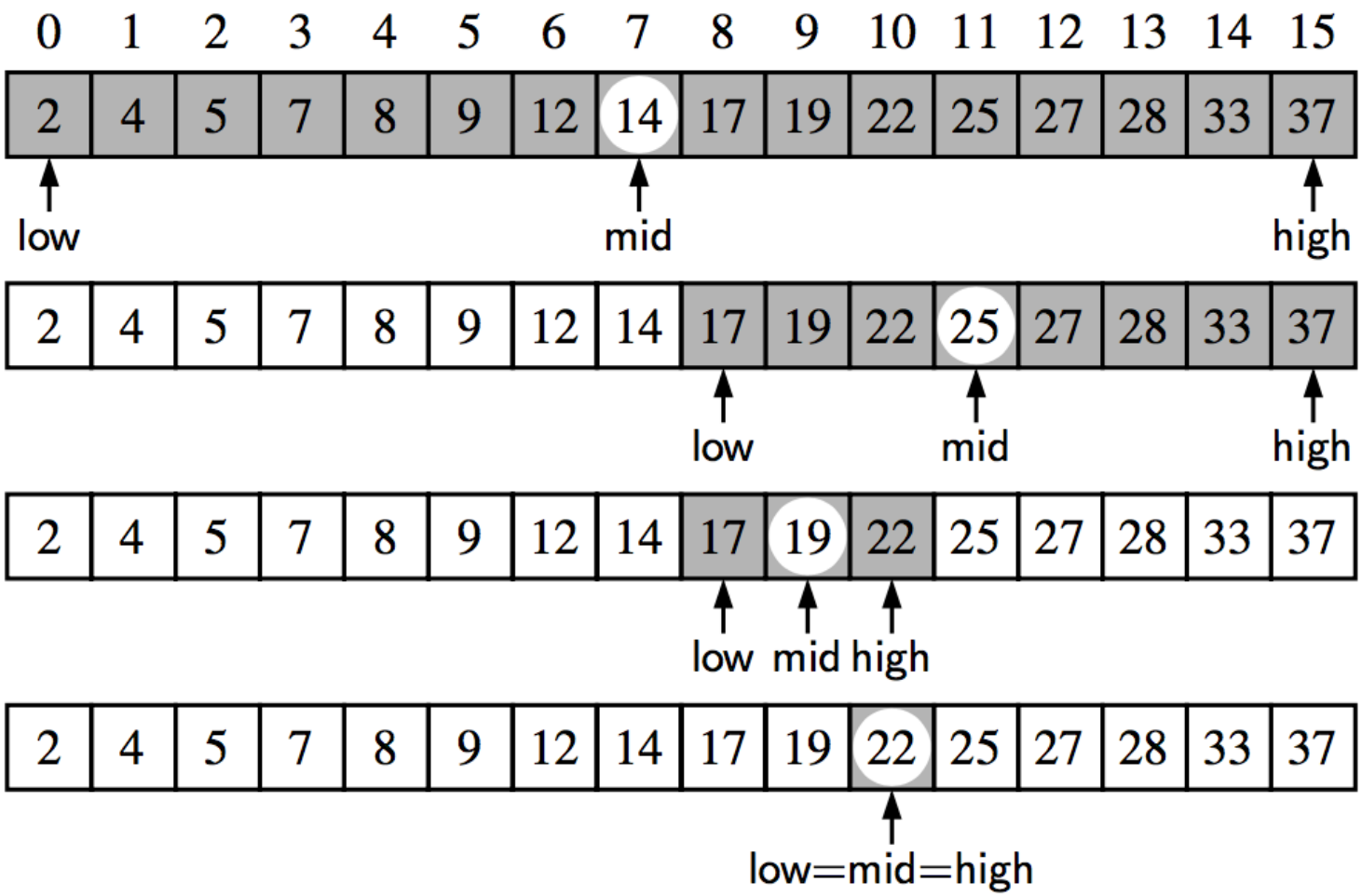

Figure 4.5: Example of a binary search for target value 22.

折半查找: Algorithm Analysis

定理

对一个长度为 n 的排序序列，折半查找算法的时间复杂度为 $O(\log n)$.

证明

由 mid 的定义知

$$(mid-1)-low+1=\left\lfloor\frac{low+high}{2}\right\rfloor-low\le\frac{high-low+1}{2}$$

或

$$high-(mid+1)+1=high-\left\lfloor\frac{low+high}{2}\right\rfloor\le\frac{high-low+1}{2}$$

故递归调用的最大次数为使得 $\frac{n}{2^r}<1$ 成立的最小整数，即 $r>\log n$.

递归的一些例子

文件系统

文件系统

现代操作系统以递归的方式定义文件系统目录。

- 一个文件系统由一个顶级目录组成，这个目录的内容由文件和其他目录组成，这些目录又可以包含文件和其他目录，等等。
- 操作系统允许目录被任意深度嵌套（只要内存中有足够的空间），尽管必须有一些只包含文件的基目录，而不是进一步的子目录。

文件系统

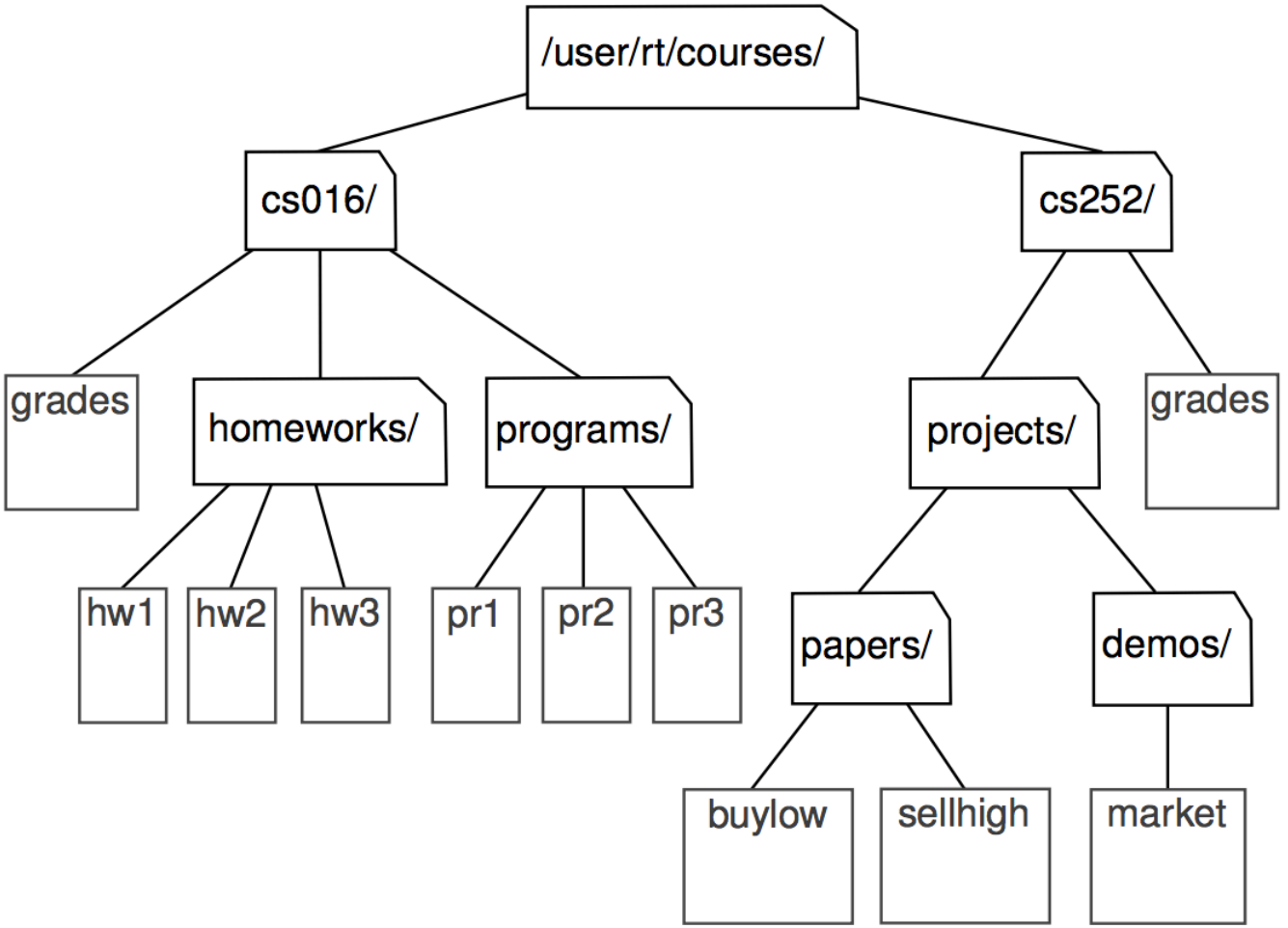

Figure 4.6: A portion of a file system demonstrating a nested organization.

文件系统

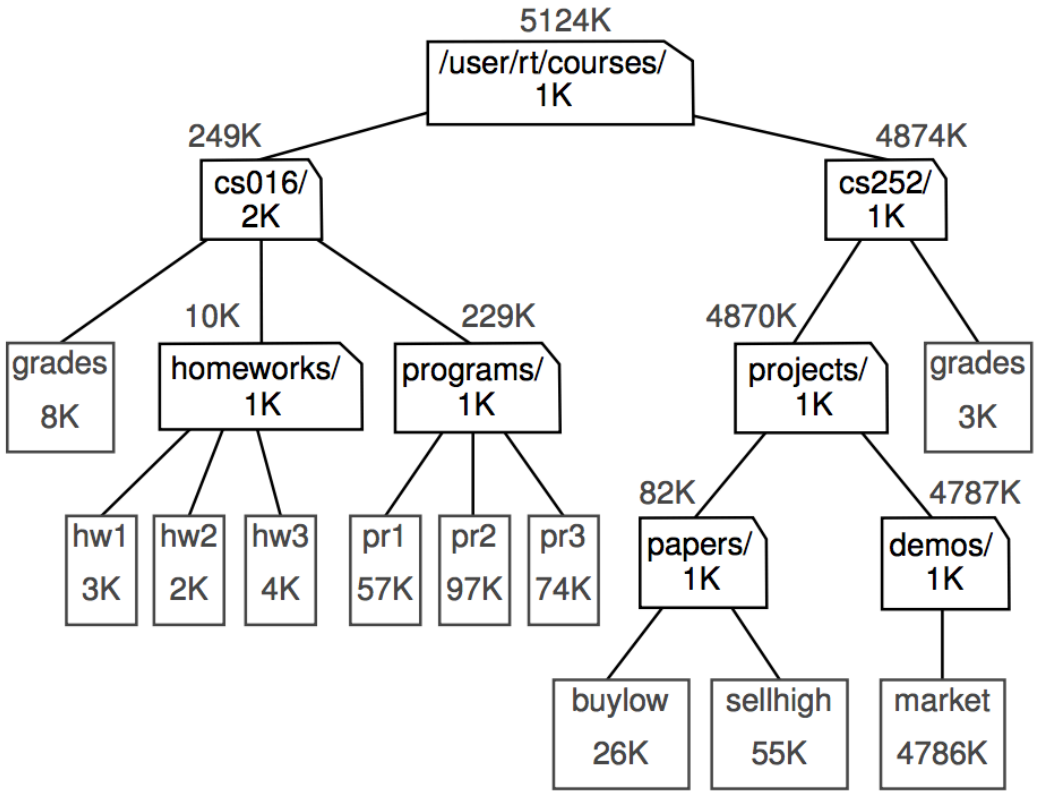

Figure 4.7: The same portion of a file system given in Figure 4.6, but with additional annotations to describe the amount of disk space that is used. Within the icon for each file or directory is the amount of space directly used by that artifact. Above the icon for each directory is an indication of the *cumulative* disk space used by that directory and all its (recursive) contents.

一个目录的累积磁盘空间 (cumulative disk space)可以用一个简单的递归算法来计算。它等于该目录使用的直接磁盘空间加上直接存储在该目录中的任何子目录的累计磁盘空间使用量之和。

文件系统

```
Algorithm DiskUsage(path):
    Input: A string designating a path to a file-system entry
    Output: The cumulative disk space used by that entry and any nested entries
    total = size(path)                    {immediate disk space used by the entry}
    if path represents a directory then
        for each child entry stored within directory path do
            total = total + DiskUsage(child)                    {recursive call}
    return total
```

Python's os Module

Python 的 os 模块，提供了在程序执行期间与操作系统交互的工具。

- `os.path.getsize(path)`

  返回由字符串路径（例如/user/rt/courses）标识的文件或目录的直接磁盘使用量（以字节为单位）。

- `os.path.isdir(path)`

  如果字符串路径指定的条目是目录，则返回 True；否则返回 False。

- `os.listdir(path)`

  返回字符串列表，该列表是由字符串路径指定的目录中所有项的名称。

- `os.path.join(path, filename)`

  使用合适的操作系统分隔符来组合路径字符串和文件名字符串。返回表示文件完整路径的字符串。

文件系统

```
import os
def disk_usage(path):
  total = os.path.getsize(path)
  if os.path.isdir(path):
    for filename in os.listdir(path):
      childpath = os.path.join(path, filename)
      total += disk_usage(childpath)
  print ('{0:<7}'.format(total), path)
  return total

if __name__ == "__main__":
  disk_usage(".")
```

低效的递归

低效的递归

- 尽管递归是一个非常强大的工具，但很容易被各种方式误用。
- 一个不好的递归实现将导致效率的极度低下。
- 应该学习一些认识和避免这种陷阱的策略。

低效的递归

元素互异问题 (Element uniqueness problem)

低效的递归

问题：元素互异

确定序列的所有 n 元素是否唯一。

低效的递归

- 若 $n = 1$, 则元素显然互异;
- 若 $n \geq 2$，则元素互异当且仅当
  - 前 $n-1$ 个元素互异
  - 后 $n-1$ 个元素互异
  - 首尾两个元素互异

元素互异问题 (Element uniqueness problem)

```
def unique3(S, start, stop):
  if stop - start <= 1:
    return True
  elif not unique(S, start, stop-1):
    return False
  elif not unique(S, start+1, stop):
    return False
  else:
    return S[start] != S[stop-1]
```

元素互异问题 (Element uniqueness problem)

令 n 为元素个数，即 $n = stop - start$.

- 当 $n = 1$ 时，没有递归调用，故unique3的时间复杂度为 $O(1)$;
- 当 $n > 1$ 时，调用一次问题规模为 n 的unique3可能导致两次问题规模为 $n-1$ 的递归调用。

元素互异问题 (Element uniqueness problem)

令 n 为元素个数，即 $n = stop - start$.

- 当 $n = 1$ 时，没有递归调用，故unique3的时间复杂度为 $O(1)$;
- 当 $n > 1$ 时，调用一次问题规模为 n 的unique3可能导致两次问题规模为 $n-1$ 的递归调用。

因此，在最坏情况下总共的函数调用为

$$1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1,$$

即总的时间复杂度为 $O(2^n)$.

低效的递归

斐波那契数列

斐波那契数列

问题：斐波那契数列

$$
\begin{aligned}
F_0 &= 0, \\
F_1 &= 1, \\
F_n &= F_{n-1} + F_{n-2}, \quad n > 1.
\end{aligned}
$$

斐波那契数列：糟糕的递归实现

```
def bad_fibonacci(n):
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n
    -1)
```

斐波那契数列：糟糕的递归实现

```
def bad_fibonacci(n):
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n
    -1)
```

按公式直接实现非常低效

斐波那契数列：糟糕的递归实现

令 c_n 为执行bad_fibonacci(n)所发生的调用次数，则

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 5 \\
c_4 &= 1 + c_2 + c_3 = 9 \\
c_5 &= 1 + c_3 + c_4 = 15 \\
c_6 &= 1 + c_4 + c_5 = 25 \\
c_7 &= 1 + c_5 + c_6 = 41 \\
c_8 &= 1 + c_6 + c_7 = 67
\end{aligned}
$$

从而有

$$c_n > 2^{n/2},$$

这意味着执行bad_fibonacci(n)的时间复杂度为 $O(2^n)$.

斐波那契数列：好的递归实现

在糟糕递归公式中，F_n 依赖于 F_{n-2} 和 F_{n-1}。但是请注意，在计算 F_{n-2} 之后，计算 F_{n-1} 的调用需要自己的递归调用来计算 F_{n-2}，因为它不知道在早期递归级别计算的 F_{n-2} 的值。这种滚雪球效应导致了糟糕递归的指数运行时间为 $O(2^n)$。

斐波那契数列: 好的递归实现

$$
\begin{aligned}
(F_0, F_{-1}) &= (1, 0), & n = 0 \\
(F_n, F_{n-1}) &= (F_{n-1} + F_{n-2}, F_{n-1}) & n > 1
\end{aligned}
$$

```
def good_fibonacci(n):
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n-1)
    return (a+b, a)
```

good_fibonacci(n)的时间复杂度为 $O(n)$.

低效的递归

Python 的最大递归深度

Python 的最大递归深度

滥用递归的另一个危险被称为无限递归。如果每个递归调用都进行另一个递归调用，而从来没有达到基本情况，那么我们就有一系列这样的调用。这是一个致命的错误。无限递归可以快速地耗尽计算资源，这不仅是因为 CPU 的快速使用，而且因为每个连续的调用都会创建一个需要额外内存的活动记录。

Python 的最大递归深度

例：Infinite recursion

```
def fib(n):
  return fib(n)
```

Python 的最大递归深度

例：Infinite recursion

```
def fib(n):
  return fib(n)
```

程序员应该确保每个递归调用都在某种程度上朝着一个基本情况发展（例如，有一个随每次调用而减少的参数值）。

Python 的最大递归深度

为避免无限递归，Python 的设计者故意限制可以同时激活的函数的总数。

- 限制次数依赖于 Python 版本，但典型的默认值为 1000；
- 若达到限制次数，Python 解释器将抛出一个 RuntimeError 异常.

Python 的最大递归深度

Python 解释器允许修改默认限制次数:

```
import sys
old = sys.getrecursionlimit()
sys.setresursionlimit(1000000)
```

递归的更多例子

递归的更多例子

线性递归

递归的更多例子

定义：线性递归

如果递归函数对主体的每次调用最多进行一次新的递归调用，这就是所谓的“线性递归”。

递归的更多例子

定义：线性递归

如果递归函数对主体的每次调用最多进行一次新的递归调用，这就是所谓的“线性递归”。

例：线性递归

- 斐波拉契数列，包括差的递归实现和好的递归实现
- 折半查找算法

  它包括一个案例分析，其中有两个分支可导致递归调用，但在主体的特定执行过程中只能访问其中一个调用。

线性递归

线性递归定义的一个结果是，任何递归跟踪都将显示为单个调用序列。

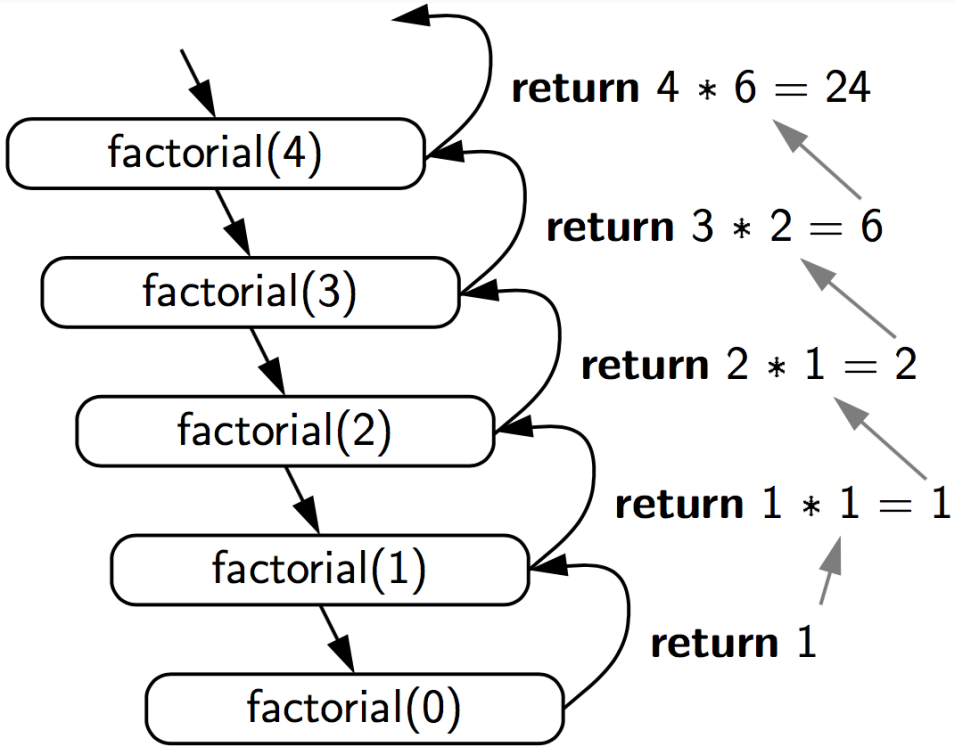

Figure 4.1: A recursion trace for the call factorial(5).

线性递归：序列求和

例：序列求和

对序列 $S = [a_1, a_2, \cdots, a_n]$，编写递归函数计算

$$s_n = \sum_{i=1}^{n} a_i$$

递归形式为

$$s_n = \begin{cases} 0 & n = 0 \\ s_{n-1} + a_n & n > 0 \end{cases}$$

线性递归：序列求和

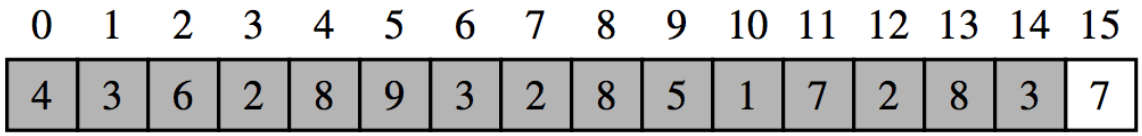

Figure 4.9: Computing the sum of a sequence recursively, by adding the last number to the sum of the first $n - 1$.

线性递归: 序列求和

```
def linear_sum(S, n):
  if n == 0:
    return 0
  else:
    return linear_sum(S, n-1) + S[n-1]
```

算法分析

- 时间复杂度
  - $n+1$ 次函数调用
  - 每次调用的时间复杂度为 $O(1)$

  总的时间复杂度为 $O(n)$
- 空间复杂度
  -

线性递归: 翻转序列 (Reversing a Sequence)

例：翻转序列

给定一个长度为 n 的序列 S，编写递归函数将其翻转。如 $S = [4,3,6,2,8,9,5]$，翻转之后变为 $[5,9,8,2,6,3,4]$.

线性递归: 翻转序列 (Reversing a Sequence)

例：翻转序列

给定一个长度为 n 的序列 S，编写递归函数将其翻转。如 $S = [4,3,6,2,8,9,5]$，翻转之后变为 $[5,9,8,2,6,3,4]$.

线性递归: 翻转序列

```
def reverse(S, start, stop):
  if start < stop - 1:
    S[start], S[stop-1] = S[stop-1], S[start]
    reverse(S, start+1, stop-1)
```

线性递归：翻转序列

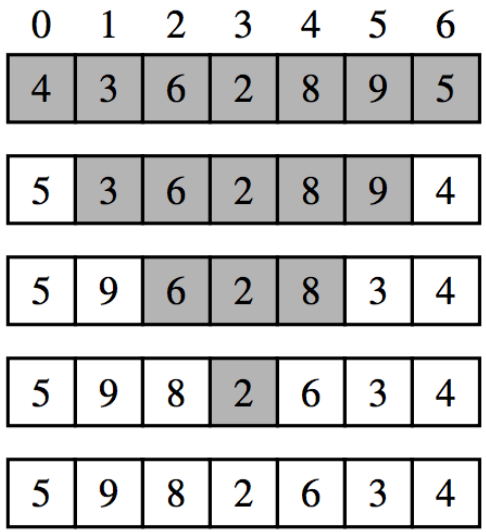

Figure 4.11: A trace of the recursion for reversing a sequence. The shaded portion has yet to be reversed.

线性递归：翻转序列

- 有两个隐含的基准情形
  - 若$start == stop$，则区间为空
  - 若$start == stop - 1$，则区间中只包含一个元素

  这两种情形都不需要进行递归调用。

- 每做一次递归调用，$stop - start$将递减 2，离基准情形更近一步。

线性递归：翻转序列

- 有两个隐含的基准情形
  - 若$start == stop$，则区间为空
  - 若$start == stop - 1$，则区间中只包含一个元素

  这两种情形都不需要进行递归调用。

- 每做一次递归调用，$stop - start$将递减 2，离基准情形更近一步。

以上分析表明该递归算法将执行 $1 + \lfloor \frac{n}{2} \rfloor$ 次递归调用，而每次调用仅包含常数量级的操作，故其时间复杂度为 $O(n)$.

线性递归: 计算幂

问题：计算幂 $power(x,n) = x^n$

- 差的递归

$$power(x,n) = \begin{cases} 1 & \text{if} \quad n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

- 好的递归

$$power(x,n) = \begin{cases} 1 & \text{if} \quad n = 0 \\ x \cdot power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)^2 & \text{if} \quad n > 0 \quad \text{is odd} \\ power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)^2 & \text{if} \quad n > 0 \quad \text{is even} \end{cases}$$

线性递归: 计算幂

```
def power_slow(x, n):
  if n == 0:
    return 1
  else:
    return x * power(x, n-1)
```

线性递归: 计算幂

```
def power_slow(x, n):
  if n == 0:
    return 1
  else:
    return x * power(x, n-1)
```

类似于$factorial(n)$，$power_slow(x,n)$的时间复杂度为 $O(n)$.

线性递归: 计算幂

```
def power_fast(x, n):
  if n == 0:
    return 1
  else:
    partial = power(x, n // 2)
    result = partial * partial
    if n % 2 == 1:
      result *= x
    return result
```

线性递归: 计算幂

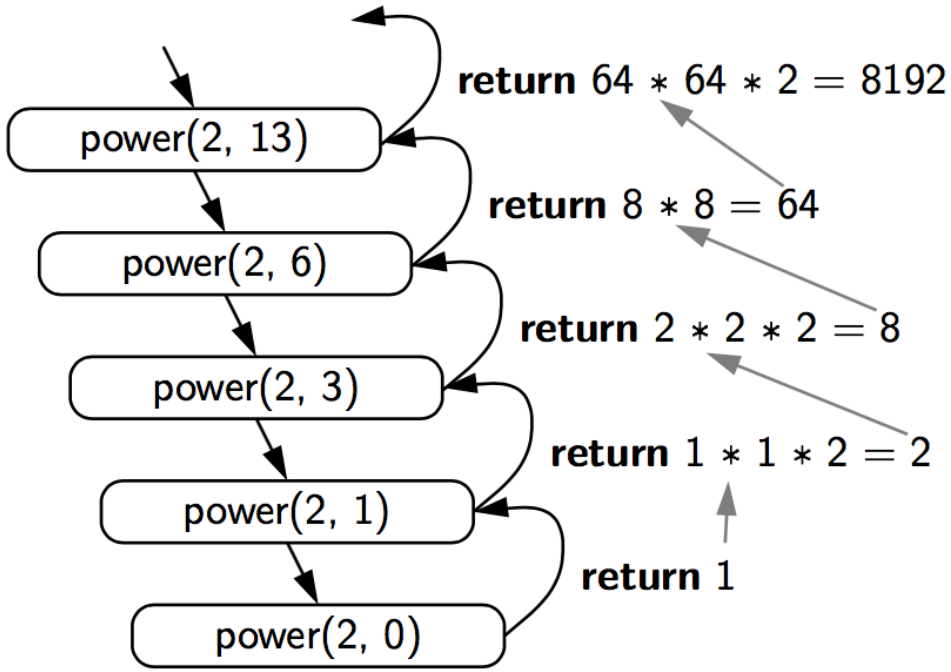

Figure 4.12: Recursion trace for an execution of power(2, 13).

线性递归: 计算幂

算法分析

- 时间复杂度
  - 递归调用次数为 $O(\log n)$
  - 每次递归调用只执行 $O(1)$ 次操作

  从而总的时间复杂度为 $O(\log n)$
- 空间复杂度
  - 因递归深度为 $O(\log n)$，故空间复杂度也为 $O(\log n)$.

递归的更多例子

二元递归 (Binary Recursion)

二元递归 (Binary Recursion)

定义：二元递归

若每次函数调用会引起两次递归调用，则称之为二元递归.

例：二元递归

- 英国尺子
- 差的斐波拉契数列

二元递归 (Binary Recursion)

例：列表求和

求长度为 n 的列表的各元素之和。

二元递归 (Binary Recursion)

例：列表求和

求长度为 n 的列表的各元素之和。

算法描述

- 求 0 个或 1 个元素之和非常容易
- 当列表有两个或两个以上元素时，可递归地计算左半部分之和与右半部分之和，然后两者相加。

二元递归 (Binary Recursion)

```
def binary_sum(S, start, stop):
  if start >= stop:
    return 0
  elif start == stop-1:
    return S[start]
  else:
    mid = (start + stop) // 2
    return binary_sum(S, start, mid) +
    binary_sum(S, mid, stop)
```

二元递归 (Binary Recursion)

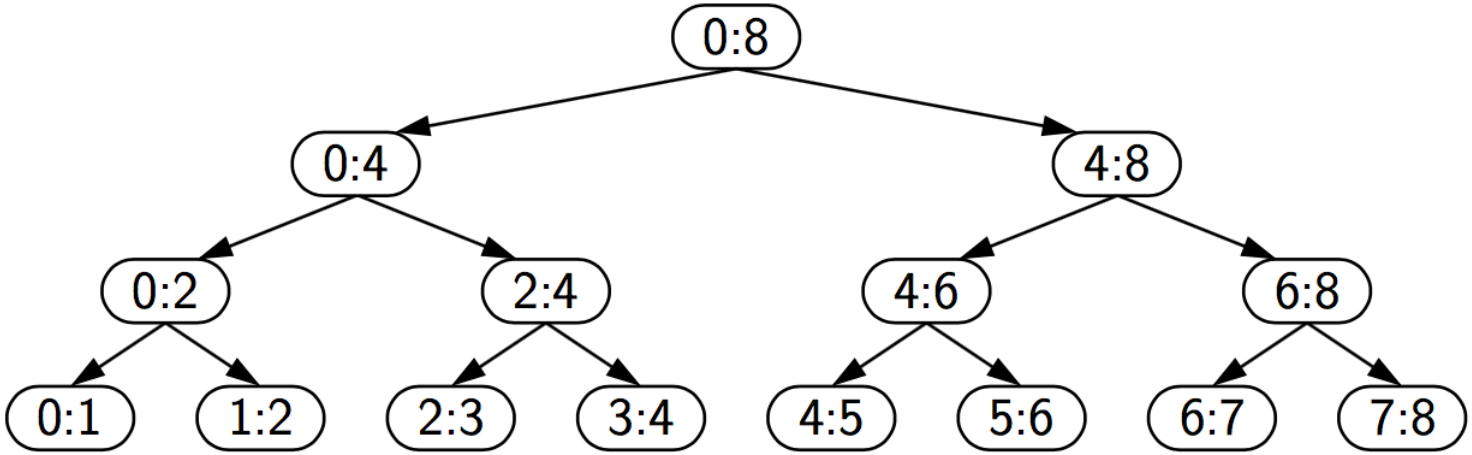

Figure 4.13: Recursion trace for the execution of binary_sum(0, 8).

二元递归 (Binary Recursion)

算法分析

为分析方便，设 $n = 2^k$.

- 时间复杂度

  因函数调用次数为 $2n-1$, 而每次调用执行 $O(1)$ 次操作, 故总的时间复杂度为 $O(n)$.

- 空间复杂度

  因递归深度为 $1+\log_2 n$，故空间复杂度为 $O(\log n)$，这相比于linear_sum函数 $O(n)$ 的空间复杂度有很大的提升.

递归的更多例子

多元递归 (Multiple Recursion)

多元递归 (Multiple Recursion)

定义：多元递归

若每次函数调用引起两次以上的递归调用，则称之为多元递归.

例：多元递归

- 计算文件系统的磁盘使用量

  递归调用的次数等于给定文件夹中的文件或文件夹个数.

设计递归算法

设计递归算法

通常，使用递归的算法通常具有以下形式：

- 基准情形

  基准情形至少要有一个，它不能包含递归调用. 应该定义这些基本情况，以便每个可能的递归调用链最终都会到达一个基本情况，并且对每个基本情况的处理不应该使用递归。

- 递归

  如果不是基本情况，则执行一个或多个递归调用。这个递归步骤可能涉及一个测试，该测试决定在几个可能的递归调用中进行哪一个。我们应该定义每个可能的递归调用，以便它朝着基本情况前进。

Parameterizing a Recursion

欲设计一个问题的递归算法，

- 需要定义子问题，并且确保它与原问题有同样的结构；
- 如果很难找到设计递归算法所需的重复结构，有时可以通过几个具体的例子来解决这个问题，看看应该如何定义子问题。

参数化递归函数

一个成功的递归设计有时需要重新定义原始问题，以便定义结构类似的子问题。一种常见的做法是递归函数的重新参数化.

例：折半查找

- 常见的函数定义为 $binary\_search(data, target)$
- 而递归函数应定义为$binary_search(data, target, low, high)$，多出两个参数来指定子列表的范围.

参数化递归函数

一个成功的递归设计有时需要重新定义原始问题，以便定义结构类似的子问题。一种常见的做法是递归函数的重新参数化.

例：折半查找

- 常见的函数定义为 $binary\_search(data, target)$
- 而递归函数应定义为$binary_search(data, target, low, high)$，多出两个参数来指定子列表的范围.

参数化递归函数

如果我们希望为折半查找这样的算法提供一个更干净的公共接口，而不需要使用额外的参数来打扰用户，那么标准的技术是使一个函数与更干净的接口一起公共使用，例如：`binary_search(data, target)`，然后让它的主体调用具有所需递归参数的非公共实用程序函数。

参数化递归函数

```
def binary_search_recur(data, target, low, high)
:
  if low > high:
    return False
  else:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      return binary_search_recur(data, target,
      low, mid - 1)
    else:
      return binary_search_recur(data, target,
      mid + 1, high)
```

参数化递归函数

```
def binary_search(data, target):
  return binary_search_recur(data, target, 0,
  len(data))
```