算法概论

第二讲：递归和分治法

薛健

Last Modified: 2018.12.9

主要内容

1 递归和数学归纳法

1.1 递归

递归 (Recursion)

- **John McCarthy** (人工智能之父)：最早认识到递归对于计算机程序设计语言的重要性，建议在 *Algol60* (Pascal、PL/I 和 C 的前身) 中增加递归特性，并自己设计了一种引入递归数据结构的语言：*Lisp*。现在，几乎所有主流的计算机程序设计语言均支持递归
- 在数学和计算机科学中，**递归**指在函数的定义中使用函数自身的方法，也常用于描述由一种 (或多种) 简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况 (用自相似的方法描述事物的过程)
- 例如，关于某人祖先的定义：
  - 某人的双亲是他的祖先 (基本情况)
  - 某人祖先的双亲同样是某人的祖先 (递归步骤)
- 像这样的定义在数学中十分常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数

例子

Example 1.1. 斐波那契数列 (Fibonacci Sequence)

- $F_0 = 0$
- $F_1 = 1$

- $F_n = F_{n-1} + F_{n-2}$

Algorithm `Fib`(n)

```
1 if n < 2 then f ← n;
2 else
3 |   f1 ← Fib(n − 1);
4 |   f2 ← Fib(n − 2);
5 |   f ← f1 + f2;
6 end
7 return f;
```

用递归来解决问题

- **递归过程的建立：**
  1. 我们已经完成了吗？如果完成了，返回结果 (如果没有这样的终止条件，递归将会永远地继续下去)
  2. 如果没有，则简化问题，解决较容易的问题，并将结果组装成原始问题的解决办法；然后返回该解决办法
- Method 99：一种更便于理解的构造方式
  - 假设要解决的问题规模为 100 (fantasy precondition)
  - 假设已经有一个名为 p99 的子程序能够解决规模为 0∼99 的问题
  - 划分不需递归的部分 (base case)
  - 构造解决问题的程序 p，写出任何不需要 p99 就可以解决的情况的代码，其他情况下在需要的时候调用 p99 (将规模太大不能直接解决的情况分解为规模在 0∼99 之间的子问题，分别用 p99 解决)

例子

Example 1.2 (二分查找的递归版本).

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1 if last < first then return −1;
2 else
3 |   mid ← (first + last)/2;
4 |   if K = E[mid] then return mid;
5 |   else if K < E[mid] then
6 |   |   return BinarySearch99[→ BinarySearchRec](E, first, mid − 1, K);
7 |   else return BinarySearch99[→ BinarySearchRec](E, mid + 1, last, K);
8 end
9 return −1;
```

课后习题

Exercise (2). 定义文件 `xx.tar.gz` 的产生方式如下：

- 以 `xx` 为文件名的文件通过 tar 和 gzip 打包压缩产生，该文件中以字符串的方式记录了一个非负整数；
- 或者以 `xx` 为名的目录通过 tar 和 gzip 打包压缩产生，该目录中包含若干 `xx.tar.gz`。

其中，`x` ∈ `[0, 9]`。现给定一个根据上述定义生成的文件 `00.tar.gz` (该文件从课程网站下载)，请确定其中包含的以 `xx` 为文件名的文件个数以及这些文件中所记录的非负整数之和。

deadline: 2018.12.01

课后习题 (提示)

- 编写递归程序求解
- 可使用任意你所熟悉的编程语言和工具
- 大多数高级语言或脚本语言提供了解 gzip 压缩包的工具

用 Python 解压缩包

```
import os, tarfile
def unpack_path_file(pathname):
    archive = tarfile.open(pathname, 'r:gz')
    for tarinfo in archive:
        archive.extract(tarinfo, os.getcwd())
    archive.close()
```

1.2 数学归纳法

数学归纳法

- **什么是证明?** (回顾): 在一个特定的公理系统中，由公理 (axioms) (和定理 (theorems)) 推导出某些命题 (proposition) 的过程
  - 证明方法: 演绎推理，构造证明，反证法，数学归纳法，...
- **数学归纳法:** 用于证明某个给定命题对于一个无限集内的所有对象成立的证明方法
- 最常见的归纳在自然数集上进行，但归纳法同样适用于更一般的无限集:
  - 给定集合是偏序集 (partially ordered set)，即集合中的部分元素对之间定义了偏序关系 (满足自反性、反对称性和传递性的二元关系)
  - 给定集合中不存在无限递减的链

在数学中，特别是序理论中，偏序集合 (简写为 poset) 是配备了偏序关系的集合。这个关系形式化了排序、顺序或排列这个集合的元素的直觉概念。这种排序不必然需要是全部的，就是说不需要但也可以保证在这个集合内的所有对象的相互可比较性。

非严格偏序 (自反偏序) 给定集合 $\mathbf{S}$，“$\leq$”是 $\mathbf{S}$ 上的二元关系，若“$\leq$”满足:

1. 自反性: $\forall x \in \mathbf{S}$ 有 $x \leq x$;
2. 反对称性: $\forall x, y \in \mathbf{S}$，$x \leq y$ 且 $y \leq x$，则 $x = y$;
3. 传递性: $\forall x, y, z \in \mathbf{S}$，$x \leq y$ 且 $y \leq z$，则 $x \leq z$

则称“$\leq$”是 $\mathbf{S}$ 上的非严格偏序或自反偏序。

严格偏序 (反自反偏序) 给定集合 $\mathbf{S}$，“$<$”是 $\mathbf{S}$ 上的二元关系，若“$<$”满足:

1. 反自反性: $\forall x \in \mathbf{S}$ 有 $x \not< x$;
2. 非对称性: $\forall x, y \in \mathbf{S}$，$x < y \Rightarrow y \not< x$;
3. 传递性: $\forall x, y, z \in \mathbf{S}$，$x < y$ 且 $y < z$，则 $x < z$

则称“$<$”是 $\mathbf{S}$ 上的严格偏序或反自反偏序。

全序关系 给定集合 $\mathbf{S}$，“$\leq$”是 $\mathbf{S}$ 上的二元关系，若“$\leq$”满足:

1. 反对称性: $\forall x, y \in \mathbf{S}$，$x \leq y$ 且 $y \leq x$，则 $x = y$;
2. 传递性: $\forall x, y, z \in \mathbf{S}$，$x \leq y$ 且 $y \leq z$，则 $x \leq z$
3. 完全性: $\forall x, y \in \mathbf{S}$，$x \leq y$ 或 $y \leq x$

数学归纳法

- **证明模式**：要证明关于 n 的语句 $F(n)$，数学归纳法可分两步：*
  1. **奠基**：当 $n=0$ 时待证语句为真，即须证 $F(0)$ 真
  2. **归纳**：在**一定的假设**下，证明情形 $n+1$ 时待证语句为真，即证明 $F(n+1)$ 真
- 完成上述两步，即可得出结论“依数学归纳法，待证语句得证”
- “**一定的假设**” —— **归纳假设**：
  - **简单归纳假设**：在假设“$F(n)$ 真”之下，去证明 $F(n+1)$ 真 $\Rightarrow$ **简单归纳证明**
  - **强归纳假设**：在假设“$F(0)$ 真，$F(1)$ 真，$\cdots$，$F(n)$ 真”之下去证明 $F(n+1)$ 真 $\Rightarrow$ **强归纳证明** ★
  - **参变归纳假设**：待证语句含有参数，如 $F(n,u)$，则奠基是：$F(0,u)$ 对一切 u 真；在归纳步骤中，假设“$F(n,u)$ 对一切 u 真”，从而证明 $F(n+1,u)$ 亦真 $\Rightarrow$ **参变归纳证明**

数学归纳法与递归

> “几乎可以说：每有一种递归式，便有一种数学归纳法；反之，每有一种数学归纳法，便有一种递归式。$\cdots$ 例如：简单归纳法对应于原始递归式，强归纳法对应于串值原始递归式，参变归纳法对应于参数变异递归式。”
>
> ——莫绍揆：《递归函数论》

- 数学归纳法与递归有着非常紧密的联系
  - 数学归纳法证明可以看作是一种递归证明
  - 数学归纳法使得递归程序正确性证明得到极大的简化

递归程序的数学归纳法证明 — 2-tree 外路径长度

Definition 1.1 (2-tree (full binary tree))**.** 2-tree 是这样一种二叉树，它是一棵空树或仅包含下列两种结点：

1. **外结点** (external node)：没有子树 (度为 0) 的结点，即叶结点
2. **内结点** (internal node)：恰有两棵子树的结点

Definition 1.2 (外路径长度 (external path length))**.** 一棵 2-tree T 的**外路径长度**是从根结点到所有外结点路径长度之和 (路径长度等于路径经过的边的数目)。其等价的递归定义：

1. 以外结点为根的 2-tree (单结点 2-tree) 其外路径长度为 0
2. 以内结点为根的 2-tree 外路径长度等于其左子树 L 的外路径长度加 L 的外结点个数加其右子树 R 的外路径长度加 R 的外结点个数

*莫绍揆：《递归函数论》

计算外路径长度

Algorithm `CalculateEPL`(T)

Input: A 2-tree T
Output: A pair $(epl, extNum)$

```
1 if T is a leaf then return (0, 1) ;
2 else
3 |   (eplL, extNumL) ← CalculateEPL(T.left);
4 |   (eplR, extNumR) ← CalculateEPL(T.right);
5 |   epl ← eplL + eplR + extNumL + extNumR;
6 |   extNum ← extNumL + extNumR;
7 |   return (epl, extNum);
8 end
```

外路径长度引理

Lemma 1.3. *T 是一棵 2-tree,e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$, 则有:*

1. *e 等于 T 的外路径长度*
2. *m 等于 T 的外结点个数*
3. *$e \geq m \lg m$*

Proof. 对 T 用数学归纳法。
奠基: T 是叶结点，根据 `CalculateEPL` 第 1 行，$e = 0, m = 1$ 满足 1 和 2，$0 \geq 1 \lg 1 = 0$ 满足 3。
归纳: 对于非叶结点 T，假设 T 的所有子树 S 引理成立，令 L 和 R 分别为 T 的左、右子树，则根据归纳假设，对于 L 和 R 引理也成立。由 `CalculateEPL` 第 3 到 7 行有:

$$\begin{aligned} e &= e_L + e_R + m_L + m_R \\ m &= m_L + m_R \end{aligned}$$

由外路径长度的递归定义可知 e 为 T 的外路径长度；又因为 T 的外结点要么在 L 中，要么在 R 中，所以 m 等于 T 的外结点个数；另外，根据归纳假设，有:

$$e \geq m_L \lg m_L + m_R \lg m_R + m$$

注意到 $x \lg x$ 是凸函数，则根据凸函数性质有:

$$m_L \lg m_L + m_R \lg m_R \geq 2\Big(\frac{m_L + m_R}{2}\Big) \lg \Big(\frac{m_L + m_R}{2}\Big)$$

所以得: $e \geq m(\lg m - 1) + m = m \lg m$ □

Corollary 1.4. *内结点数为 n 的 2-tree 外路径长度 $e \geq (n+1)\lg(n+1)$*

1.3 程序正确性证明

定义和术语

Definition 1.5.

程序块 (block) 一段有且仅有一个入口和一个出口的程序代码

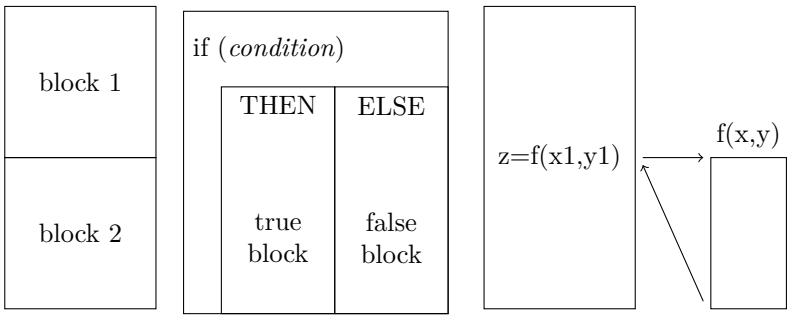
过程 (procedure) 赋予名称且可被调用的程序块，一般包含输入和输出参数

函数 (function) 有输出参数的过程

前件 (precondition) 用于描述一个程序块的输入参数和非局部数据且在进入程序块时为真的逻辑语句

后件 (postcondition) 用于描述一个程序块的输入参数、输出参数及非局部数据且在退出程序块时为真的逻辑语句

基本控制结构



循环结构 (**for**, **while**, ...) $\Longrightarrow$ 递归 !

基本证明形式

Proposition 1.6 (General correctness lemma form)**.** 如果在进入程序块时前件成立，则在退出程序块时后件也成立。

Proposition 1.7 (Sequence correctness lemma form)**.**

1. 整个程序块的前件 $\Rightarrow$ 程序块 *1* 的前件
2. 程序块 *1* 的后件 $\Rightarrow$ 程序块 *2* 的前件
3. 程序块 *2* 的后件 $\Rightarrow$ 整个程序块的后件

Proposition 1.8 (Procedure-call correctness lemma form)**.**

1. 程序块的前件 $\Rightarrow$ 被调用过程的前件及其实际调用参数
2. 被调用过程的后件及其实际调用参数 $\Rightarrow$ 程序块的后件

Proposition 1.9 (Alternation correctness lemma form)**.**

1. 程序块的前件以及选择分支条件为真 $\Rightarrow$ 条件为真时的分支程序块前件
2. 条件为真时的分支程序块后件以及选择分支条件为真 $\Rightarrow$ 程序块的后件
3. 程序块的前件以及选择分支条件为假 $\Rightarrow$ 条件为假时的分支程序块前件
4. 条件为假时的分支程序块后件以及选择分支条件为假 $\Rightarrow$ 程序块的后件

单赋值形式

- 证明的困难：到逻辑表达式的转换
  - 无条件跳转 (goto) 语句：可以消除
  - 赋值 (assignment) 语句：“y=y+1”，“x=y+1; y=z;”，...
- **单赋值形式：**Single-Assignment Paradigm 或 Static Single Assignment Form (SSA)，如果在某个过程内赋值的每一个变量作为赋值目标只出现一次，称这个过程是 (静态) 单赋值形式。
- 在编译器设计中，单赋值形式是一种中间表示 (IR)，它能有效地将程序中的运算值和它们的存储位置分开，从而使得若干优化能具有更有效的形式
- 使用单赋值形式的语言 (编译器)：Prolog, ML, Haskell, SISAL (Streams and Iteration in a Single Assignment Language), SAC (Single Assignment C)

例子

a code fragment

```
1 if y < 0 then
2 |   y = 0;
3 end
4 x = 2 * y;
```

$\Rightarrow$

single assignment version

```
1 if y < 0 then
2 |   y1 = 0;
3 else
4 |   y1 = y;
5 end
6 x = 2 * y1;
```

$$(y < 0 \Rightarrow y1 = 0) \wedge (y \geq 0 \Rightarrow y1 = y) \wedge (x = 2 * y1)$$

无循环过程

Example 1.3 (顺序查找的递归版本).

Algorithm `SeqSearchRec`$(E[], m, num, K)$

```
1 if m ≥ num then
2 |   ans ← −1;
3 else if E[m] = K then
4 |   ans ← m;
5 else
6 |   ans ← SeqSearchRec(E, m + 1, num, K);
7 end
8 return ans;
```

循环转换成递归

准备工作：

1. 将循环内部的局部变量转换为单赋值形式
2. 对所有在循环内需要更新的变量 (通常是定义在循环外的)，将所有更新工作放到循环体末尾进行

转换步骤：

1. 循环体内更新的变量变为递归过程的输入参数，其初始值对应于顶层调用递归过程时实际传入的参数值 (active parameters)
2. 在循环外定义但在循环内只访问不更新的变量也变为递归过程的输入参数，在递归调用过程中其值从不改变 (passive parameters)
3. 递归过程起始处检测循环条件，若为假则返回结果 (退出)，若为真则执行循环体内容；循环体中的 break 语句也对应于递归过程的返回
4. 当转换至循环体结束时，调用递归过程本身，其实际调用参数即为循环内更新后的变量值

例子

Example 1.4 (阶乘函数).

Algorithm FactLoop(n)

```
1 k ← 1;
2 f ← 1;
3 while k ≤ n do
4 |   fnew ← f * k;
5 |   knew ← k + 1;
6 |   k ← knew;
7 |   f ← fnew;
8 end
9 return f;
```

转换结果

Example 1.5 (阶乘函数的递归版本).

Algorithm Fact(n)

```
1 return FactRec(n, 1, 1);
```

Algorithm FactRec(n, k, f)

```
1 if k > n then ans ← f;
2 else
3 |   fnew ← f * k;
4 |   knew ← k + 1;
5 |   ans ← FactRec(n, knew, fnew);
6 end
7 return ans;
```

证明实例

Example 1.6 (二分查找的规范化递归版本).

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1 if last < first then index ← −1;
2 else
3 |   mid ← (first + last)/2;
4 |   if K = E[mid] then index ← mid;
5 |   else if K < E[mid] then
6 |   |   index ← BinarySearchRec(E, first, mid − 1, K);
7 |   else index ← BinarySearchRec(E, mid + 1, last, K);
8 end
9 return index;
```

证明

Lemma 1.10. 当 `BinarySearchRec`$(E, first, last, K)$ 被调用，其问题规模为 $last - first + 1 = n$,且 $E[first], \cdots, E[last]$ 以不减序排列,则对任意 $n \geq 0$,如果 K 不在 $E[first], \cdots, E[last]$ 中，过程调用返回 -1，否则返回 $index$ 满足 $K = E[index]$.

Proof. 对问题规模 n 归纳：
奠基：当 $n = 0$ 时，$last = first - 1$，第 1 行条件满足，返回值为 -1，结论成立；
归纳：当 $n > 0$ 时，假设对于 $0 \leq k < n$，`BinarySearchRec`(E, f, l, K) 调用结论成立，其中 $l - f + 1 = k$。则当 $k = n$ 时，因为 $n > 0$，所以第 1 行条件为假，程序执行到第 3 行，这时 $mid = \lfloor (first + last)/2 \rfloor$，所以 $first \leq mid \leq last$。若第 4 行条件为真，则返回 $index = mid$ 为 K 在序列 $E[first], \cdots, E[last]$ 中的位置，结论成立；
若第 4 行条件为假，由上述条件可知

$$(mid - 1) - first + 1 < last - first + 1 = n$$
$$last - (mid + 1) + 1 < last - first + 1 = n$$

因此第 6、7 行的递归调用满足归纳假设，可以返回正确结果。
当第 5 行条件为真时，第 6 行递归调用被执行，若返回非负整数值，则问题解决，结论成立；若返回 -1，则表示 K 不在 $E[first], \cdots, E[mid - 1]$ 中，而第 5 行条件为真表示 K 也不在 $E[mid], \cdots, E[last]$ 中，因此返回 -1 对当前调用也是正确的。
与之类似，可证明第 5 行条件为假时结论也成立。 □

1.4 递推方程

递推方程

- **递推方程** (Recurrence Equation)：一种递推地定义一个序列的方程式：序列的每一项定义为前一项的函数。
- 递推方程可以很自然地用来描述递归程序执行过程的资源 (时间、空间...) 使用情况 (统一称其为“开销 (cost) ”)
- 最坏情况下程序开销递推方程的建立 $T(n) = ?$：
  1. 对于顺序执行的程序块，直接加上它的开销；
  2. 对于有选择分支的程序块 (非递归终止分支，nonbase cases)，加上分支中最大开销；
  3. 对于子程序调用，加上所调用子程序的开销 $T_s(n_s(n))$，其中 $n_s(n)$ 为所调用子程序的输入规模，是主程序输入规模 n 的函数；
  4. 对于递归调用，加上递归调用开销 $T(n_r(n))$，其中 $n_r(n)$ 为递归调用的规模，是主程序输入规模 n 的函数
- 如何求 $T(n)$?

例子

- 递归版的顺序查找：

$$\big(0 + (1 + \max(0, T(n-1))\big) + 0$$

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

- 递归版的二分查找：

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

常用递推方程形式

- Divide and Conquer：规模为 n 的问题被分解为 b 个规模为 n/c $(c > 1)$ 的子问题，$f(n)$ 为非递归部分 (分解和合并) 的开销，$b \geq 1$ 称为分支因子 (branching factor)

$$T(n) = bT(n/c) + f(n)$$

- Chip and Conquer：规模为 n 的问题被“裁剪”成规模为 $n - c$ 的子问题

$$T(n) = T(n-c) + f(n)$$

- Chip and Be Conquered：规模为 n 的问题被分解并“裁剪”成 b 个规模为 $n - c$ 的子问题

$$T(n) = bT(n-c) + f(n)$$

递推方程求解

Example 1.7 $(T(n) = 2T(n/2) + n \lg n)$.

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n\lg n = 2(2T(\frac{n}{4}) + \frac{n}{2}\lg\frac{n}{2}) + n\lg n \\
&= 2^2T(\frac{n}{2^2}) + n\lg\frac{n}{2} + n\lg n \\
&= 2^3T(\frac{n}{2^3}) + n\lg\frac{n}{2^2} + n\lg\frac{n}{2} + n\lg n \\
&= \cdots \\
&= 2^dT(\frac{n}{2^d}) + n(\lg\frac{n}{2^{d-1}} + \cdots + \lg\frac{n}{2^0}) \\
&= nT(1) + n(\lg n - (d-1) + \lg n - (d-2) + \cdots + \lg n - 0) \\
&= nT(1) + nd\lg n - n\frac{d(d-1)}{2} \\
&= nT(1) + \frac{n}{2}\lg^2 n + \frac{n}{2}\lg n \in \Theta(n\lg^2 n)
\end{aligned}$$

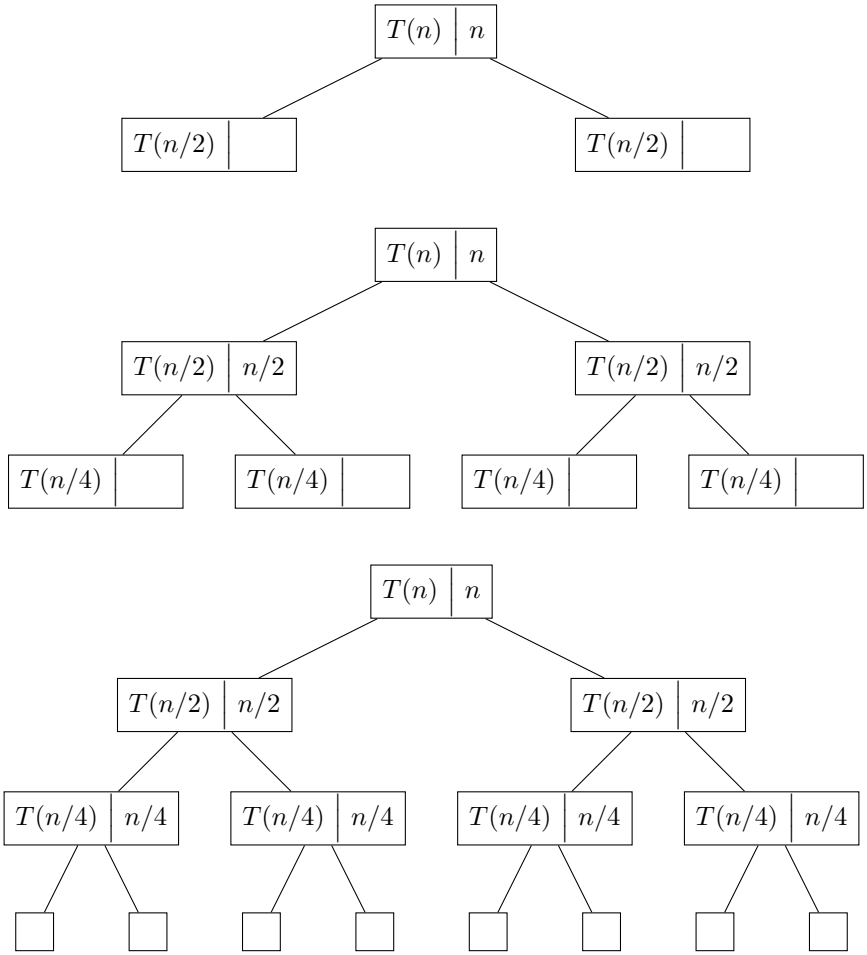
递归树 (Recursion Tree)

Divide and Conquer

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

$T(size)$	$nonrec.\ cost$

$T(n)$	



$$n/2^d = 1 \Rightarrow T(n) \approx n \lg n$$

Divid and Conquer 一般情况

$$T(n) = bT(n/c) + f(n)$$

定义 $E = \log_c b = \dfrac{\lg b}{\lg c}$ 称为关键指数 (critical exponent)

Lemma 1.11. *Divid and Conquer* 递归树的叶结点个数 $L \approx n^E$

Lemma 1.12.

1. 递归树的深度 (高度) $D \approx \lg n / \lg c = \log_c n$
2. 第 0 层的开销和为 $f(n)$
3. 假设递归基础的开销为 1，第 D 层的开销和为 n^E
4. $T(n)$ 等于递归树中所有结点的非递归开销总和，也就是递归树每一层“开销和”的总和

主定理

Theorem 1.13 (Little Master Theorem)**.** 对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若递归树每一层的开销和呈递增几何级数，则 $T(n) \in \Theta(n^E)$

2. 若递归树每一层的开销和保持不变，则 $T(n) \in \Theta(f(n)\log n)$

3. 若递归树每一层的开销和呈递减几何级数，则 $T(n) \in \Theta(f(n))$

Theorem 1.14 (Master Theorem)**.** 对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若对常数 $\epsilon > 0$，$f(n) \in O(n^{E-\epsilon})$，则 $T(n) \in \Theta(n^E)$

2. 若 $f(n) \in \Theta(n^E)$，则 $T(n) \in \Theta(n^E \log n)$
   更一般地：若 $f(n) \in \Theta(n^E \log^k n)$，则 $T(n) \in \Theta(n^E \log^{k+1} n)$

3. 若对常数 $\epsilon > 0$，$f(n) \in \Omega(n^{E+\epsilon})$，并且存在常数 $\delta < 1$ 以及整数 n_0，使得当 $n > n_0$ 时有 $bf(n/c) \le \delta f(n)$，则 $T(n) \in \Theta(f(n))$

主定理的证明

证明要点. 递归树中在第 d 层 (深度为 d) 有 b^d 个结点，每个非叶结点的非递归开销为 $f(n/c^d)$，因此有

$$T(n) = n^{\log_c b}\Theta(1) + \sum_{d=0}^{\log_c n - 1} b^d f\left(\frac{n}{c^d}\right)$$

令 $g(n) = \sum_{d=0}^{\log_c n - 1} b^d f\left(\frac{n}{c^d}\right)$，则：

1. 在条件 1 下，有 $g(n) \in O(n^E)$

2. 在条件 2 下，有 $g(n) \in \Theta(n^E \log n)$

3. 在条件 3 下，有 $g(n) \in \Theta(f(n))$

□

主定理的应用

Example 1.8 ($T(n) = 9T(n/3) + n$). $\log_c b = \log_3 9 = 2$，取 $\epsilon = 0.5$，则有 $f(n) = n \in O(n^{2-0.5}) = O(2^{1.5})$，
所以，$T(n) \in \Theta(n^2)$

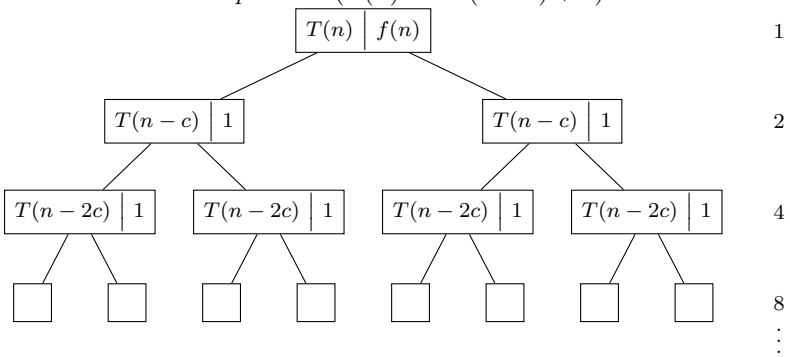
Example 1.9 ($T(n) = T(2n/3) + 1$). $\log_c b = \log_{3/2} 1 = 0$，故 $f(n) = 1 \in \Theta(n^0)$，
所以 $T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$

Example 1.10 ($T(n) = 3T(n/4) + n\lg n$). $\log_c b = \log_4 3 < 1$，故存在 $\epsilon > 0$ 使 $\log_4 3 + \epsilon < 1$，则 $f(n) = n\lg n \in \Omega(n) \subset \Omega(n^{\log_4 3+\epsilon})$；
另外，$bf(n/c) = 3(n/4)\lg(n/4) = \frac{3n}{4}(\lg n - 2) \le \frac{3}{4}f(n)$，
所以 $T(n) \in \Theta(n\lg n)$

递归树

Chip and Conquer (Be Conquered)

Example 1.11 ($T(n) = 2T(n-c) + 1$).



Chip and Conquer (Be Conquered)

- $T(n) = bT(n-c) + f(n)$
- 递归树深度大约为 $d = n/c$
- $T(n) \doteq \sum_{d=0}^{n/c} b^d f(n-cd) = b^{n/c} \sum_{h=0}^{n/c} \frac{f(ch)}{b^h} = b^{n/c} g(n), \quad h = (n/c) - d$
- 在大多数实际情况下，$g(n) \in \Theta(1)$，这时 $T(n) \in \Theta(b^{n/c})$，是一个指数阶的函数！
- 若 $b = 1$：$T(n) \doteq \sum_{h=0}^{n/c} f(ch) \approx \frac{1}{c} \int_0^n f(x)dx$
  - $f(n) = n^\alpha$：$T(n) \in \Theta(n^{\alpha+1})$
  - $f(n) = \log n$：$T(n) \in \Theta(n \log n)$

2 分治法

2.1 基本原理

分治法 (Divide and Conquer)

- **工作原理**：“解决几个小问题通常比解决一个大问题来得简单”
- **设计思想**：将一个直接难以解决的大问题分成较小规模的数个子问题；分别解决 (治) 这些子问题；将子问题结果合成原问题的结果
  - 由分治法产生的子问题往往是原问题的较小模式，在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解，这自然导致递归过程的产生
- **适用条件**：
  1. 问题的规模缩小到一定的程度就可以容易地解决；
  2. 问题可以分解为若干个规模较小的相同问题；
  3. 利用该问题分解出的子问题的解可以合并为该问题的解；
  4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题 (效率)
- 使用分治法的例子：`BinarySearchRec`

算法基本结构

Algorithm `Solve`(I)

```
n ← Size(I);
if n ≤ smallSize then
    solution ← DirectlySolve(I);
else
    {I_1, ··· , I_k} ← Divide(I);
    foreach i ∈ {1, ··· , k} do S_i ← Solve(I_i) ;
    solution ← Combine(S_1, ··· , S_k);
end
return solution
```

时间复杂度：$\begin{cases} T(n) = D(n) + \sum_{i=1}^{k} T(\texttt{Size}(I_i)) + C(n) & n > smallSize \\ T(n) \in \Theta(1) & n \leq smallSize \end{cases}$

2.2 矩阵相乘

斯特拉森矩阵乘法

- 由德国数学家 Volker Strassen 于 1969 年提出，被称为 Strassen Algorithm
- A 和 B 是两个 $n \times n$ 矩阵，A 和 B 相加时间复杂度为 $\Theta(n^2)$，A 和 B 相乘时间复杂度为 $\Theta(n^3)$
- 分治法：将矩阵分成大小为 $\frac{n}{2} \times \frac{n}{2}$ 的子矩阵 (假设 $n = 2^k$)

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

- $T(n) = 8T(n/2) + c_1 n^2, \quad T(n) = c_2 \ (n \leq 2) \Rightarrow T(n) \in \Theta(n^3)$
- 在分治法基础上通过增加加法次数来减少乘法次数

$$AB = \begin{bmatrix} P + S - T + V & R + T \\ Q + S & P + R - Q + U \end{bmatrix}$$

其中：

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

- $T(n) = 7T(n/2) + c_1 n^2, \quad T(n) = c_2 \ (n \leq 2) \Rightarrow T(n) \in \Theta(n^{2.807})$

2.3 大整数乘法

问题描述

- 在分析算法复杂度时经常将乘法作为基本运算处理，即认为其时间复杂度为 $\Theta(1)$
- 但这样的分析有一个前提，就是计算机字长允许直接表示和处理参与运算的整数

- 在某些情况下需要处理很大的整数，超过了计算机字长表示范围
- 用浮点数运算限制了计算精度和有效数字位数
- 因此，对于大整数相乘，只能以软件的方式设计算法来实现
- **目标**：设计一个有效的算法进行两个 n 位大整数的乘法运算

设计思路

- 采用列竖式笔算的方法，若将每 2 个 1 位数的相乘作为基本运算单位，其时间复杂度为 $O(n^2)$
- 分治法思想：
  - 将 2 个 n 位 b 进制整数各分为两段表示如下：(为便于分析，不妨设 $n = 2^k$)

$$X = Ab^{n/2} + B, \quad Y = Cb^{n/2} + D$$

  - 则 $XY = ACb^n + (AD + BC)b^{n/2} + BD$
  - 递归开销为 4 次 $n/2$ 位整数的乘法，非递归开销为 3 次不超过 n 位整数的加法 $(O(n))$、2 次数组移位 $(O(n))$
  - 时间复杂度：$W(n) = 4W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^2)$
  - 继续改进：$XY = ACb^n + ((A - B)(D - C) + AC + BD)b^{n/2} + BD$
  - 递归开销为 3 次 $n/2$ 位整数的乘法，非递归开销为 6 次加减法和 2 次移位
  - 时间复杂度：$W(n) = 3W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^{\lg 3}) \in O(n^{1.59})$

算法描述

- 假定整数保存在大小为 $n+1 (n > 0)$ 的数组 A 中，$A[0]$ 为符号位，取 $+1$ 或 -1，$A[i]$ 对应第 i 位数字

Algorithm `LargeMul`$(X[], Y[], n)$

```
1  S ← Array(2 * n + 1, 0);
2  S[0] ← X[0] * Y[0];
3  if n = 1 then
4  |  S ← Mul(X[1], Y[1]);
5  else
6  |  mid ← n div 2;
7  |  A ← X[mid + 1..n];  B ← X[1..mid];
8  |  C ← Y[mid + 1..n];  D ← Y[1..mid];
9  |  m1 ← LargeMul(A, C, n − mid);
10 |  m2 ← LargeMul(Sub(A, B), Sub(D, C), Max(mid, n − mid));
11 |  S ← LargeMul(B, D, mid);
12 |  S ← Add(S, ShiftLeft(m2, n − mid));
13 |  S ← Add(S, ShiftLeft(m1, 2n − 2mid));
14 end
```

2.4 最接近点对问题

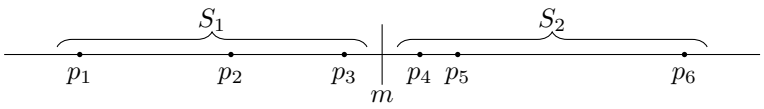
问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大

- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 ——**最接近点对问题**[†]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小
- 解决方案：
  - 两两计算点对距离，找出其中的最小值，复杂度为 $O(n^2)$
  - 可以证明该问题的复杂度下界为 $\Omega(n\log n)$
  - 是否存在复杂度为 $\Theta(n\log n)$ 的算法？ ——分治法？

分治法思路

- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- **关键问题**：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对
- 一维的情况：



  - 假定分割点为 m，则如果出现最小点对分别落在 S_1 和 S_2 中的情况，这两点只可能是 S_1 中的最大点和 S_2 中的最小点 $(O(n))$
  - 分割点 m 的选择：中位数 $(\Theta(n))$
  - 时间复杂度：$W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n\log n)$

一维情况算法描述

Algorithm `MinPairI`(S)

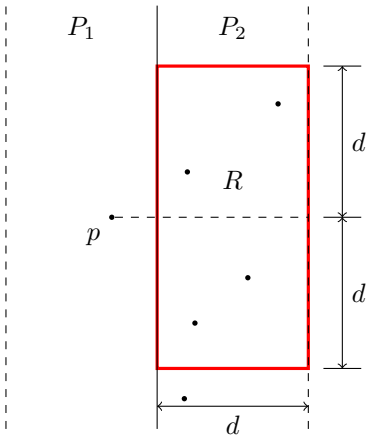
```
1  if |S| < 2 then d ← ∞ ;
2  else
3      m ← S 中点坐标中位数;
4      S1 ← {x ∈ S|x ≤ m};
5      S2 ← {x ∈ S|x > m};
6      d1 ← MinPairI(S1);
7      d2 ← MinPairI(S2);
8      x1 ← max(S1);
9      x2 ← min(S2);
10     d ← min(d1, d2, x2 − x1);
11 end
12 return d;
```

[†]王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？
  - p, q 必然位于直线 l 两侧左右各宽 d 的垂直长条内，设为 P_1、P_2
  - 对于 P_1 中任一候选点，P_2 中能与其构成候选点对的点必然落在一个 $d \times 2d$ 的矩形 R 中
  - 矩形 R 中最多只有 6 个点 (鸽笼原理)
  - 如何在线性时间复杂度内找到这 6 个点？—— 对 y 坐标预排序

示意图



算法描述

Algorithm `MinPairIIRec`(P)

```
1  if |S| < 2 then d ← ∞ ;
2  else
3      m ← P 中点的 x 坐标中位数;
4      P1[] ← {P[i]_x ≤ m};
5      P2[] ← {P[i]_x > m};
6      d1 ← MinPairIIRec(P1);
7      d2 ← MinPairIIRec(P2);
8      d ← dm ← min(d1, d2);
9      P1*[] ← {m − P1[i]_x < dm};
10     P2*[] ← {P2[i]_x − m < dm};
11     foreach P1*[i] do
12         搜索 P2* 中与 P1*[i] 距离小于 dm 的点 (最多 6 个) 更新 d;
13     end
14 end
15 return d;
```

算法分析

- 3, 4, 5, 9, 10 显然复杂度为 $O(n)$
- 8 的开销是常数
- 6, 7 是递归部分，复杂度为 $2W(n/2)$
- 关键是 11～13，在 P 是有序的前提下才能够达到线性复杂度，而且在递归调用的任何时候都不会破坏子集中的点 y 坐标的有序性
- 在上述条件下，`MinPairIIRec` 的时间复杂度：$W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n \log n)$
- 因此在调用 `MinPairIIRec` 之前需要对 P 中的点按 y 坐标排序，排序的时间复杂度可以达到 $\Theta(n \log n)$，所以不影响整个算法的时间复杂度

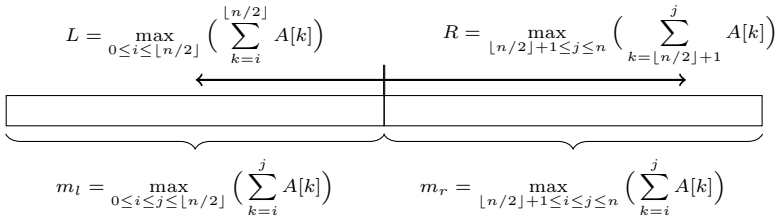
Procedure `MinPairII`(S)

```
1 P ← S 中的点按 y 坐标排序;
2 MinPairIIRec(P);
```

2.5 股价增值问题

问题描述及分析

- 某公司过去 n 天的股票价格波动保存在数组 A 中，求在哪段时间 (连续哪几天) 其股价的累计增长最大
- 例如：过去 7 天内其股价波动序列为 $+3, -6, +5, +2, -3, +4, -4$，累计增长最大值出现在第 3 天到第 6 天，累计增长值为 $5 + 2 - 3 + 4 = 8$
- 实际上就是求 i 和 j $(0 \le i \le j \le n-1)$，使 $\sum_{k=i}^{j} A[k]$ 最大
- 采用分治法设计算法：



$$m = \max(m_l, m_r, L + R)$$

算法描述

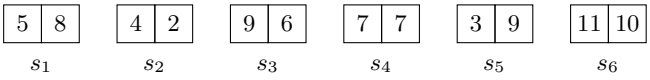
Algorithm `LargestIncrease`$(A[], first, last)$

```
1  if first ≥ last then i ← first, j ← last, m ← A[last] ;
2  else
3      mid ← ⌊(first + last)/2⌋;
4      (il, jl, ml) ← LargestIncrease(A, first, mid);
5      (ir, jr, mr) ← LargestIncrease(A, mid + 1, last);
6      找到 im 使得 L = Σ_{k=im}^{mid} A[k] 最大;
7      找到 jm 使得 R = Σ_{k=mid+1}^{jm} A[k] 最大;
8      if ml ≥ mr and ml ≥ L + R then i ← il, j ← jl, m ← ml ;
9      else if mr ≥ ml and mr ≥ L + R then i ← ir, j ← jr, m ← mr ;
10     else i ← im, j ← jm, m ← L + R ;
11 end
12 return (i, j, m);
```

- 算法复杂度：$W(n) = 2W(n/2) + \Theta(n) \Rightarrow \Theta(n \log n)$

课后习题：多米诺骨牌

Exercise (3). 现有 n 块“多米诺骨牌” $s_1, s_2, \cdots, s_n$ 水平放成一排，每块骨牌 s_i 包含左右两个部分，每个部分赋予一个非负整数值，如下图所示为包含 6 块骨牌的序列。骨牌可做 180 度旋转，使得原来在左边的值变到右边，而原来在右边的值移到左边，假设不论 s_i 如何旋转，$L[i]$ 总是存储 s_i 左边的值，$R[i]$ 总是存储 s_i 右边的值，$W[i]$ 用于存储 s_i 的状态：当 $L[i] \leq R[i]$ 时记为 0，否则记为 1，试采用分治法设计算法求 $\sum_{i=1}^{n-1} R[i] \cdot L[i+1]$ 的最大值，以及当取得最大值时每个骨牌的状态。下面是 $n = 6$ 时的一个例子：



deadline: 2018.12.08