# 介绍

## 本书讨论内容

很多问题都可以通过算法解决，但其中也有优劣问题

非常重要的一个理念：程序能够在输入数据较大时，依然良好执行

从N个数中找出第K大的数，即每次从一组数中取最大值的数，第k次取值即符合条件

字谜游戏

## 数学知识复习

指数、对数、数列、取模运算、证明方法（归纳法、反证法）

## 递归简介

典型应用斐波那契数列

调用自身、有结束条件

## java5之前实现泛型

使用Object来泛化、基本类型包装类、使用接口来表泛型、协变数组类型

B extends A。List<A>中不能添加B。A[] arr = new B[2]。C extends A。arr[0] = C。编译不报错，但运行时异常。

## java5实现泛型组件

简单泛型类、接口；自动装箱、拆箱；菱形操作符（简化泛型写法）；带有边界通配符；

泛型静态方法，泛型必须声明在方法返回值前；泛型边界；泛型擦除；泛型限制（泛型数组、泛型实例）

## 函数对象

Comparator，此接口只有一个方法声明

## 练习

练习7：

第一个证明，显而易见x<1时，明显成立；

[1,2]之间成立作为基准条件；

需要找到一个推进条件，当p<=x<2p时，等式成立；证明2p<=y<4p时，等式也成立；

log2y<y，log2(2x)<2x，1+logx<2x，1<x，如此推进条件满足。得证

第二个证明，令A=2x即可得证

练习8：

a：4/3；b：4/9；c：20/27；递归求解可得

练习9：ln2

练习10：1

练习11：

# 算法分析

## 数学背景

T(N)=O(f(N))，存在c、n0使得T(N)<=cf(N)，当N>=n0

T(N)=Ω(f(N))，存在c、n0使得T(N)>=cf(N)，当N>=n0

T(N)=Θ(f(N))，当且仅当T(N)=O(f(N))，T(N)=Ω(f(N))

T(N)=o(f(N))，存在c、n0使得T(N)<cf(N)。T(N)=O(f(N))，但T(N)≠Θ(f(N))

推导

1. T1(N)=O(f(N))，T2(N)=O(g(N))；

T1(N)+T2(N)=O(f(N)+g(N))。

T1(N)\*T2(N)=O(f(N)\*g(N))。

1. 如果T(N)是一个k阶多项式，则T(N)= Θ(Nk)
2. logkN=O(N)

## 模型

假设内存足够大、加减乘这种操作耗时一致

## 分析什么

分析运行时间

## 运行时间计算

数列的子序列求和，分析最大子序列和

N3、N2、NlogN、N为4种算法时间复杂度

为O(logN)的几种例子：二分查找法、欧几里法求最大公约数、某一数字求其k次方

## 练习

1，2/N、37、N1/2、N、NloglogN、NlogN、Nlog(N2)、Nlog2N、N2logN、N3/2、N2、N3、2N/2、2N

# 链表、栈、队列

## 抽象数据类型

一系列对象和操作的集合

## List abstract data type

1. 常见操作增、删、查，遍历
2. 数组方式实现List，查找容易，插入、删除不适合
3. 链表方式实现List，

# 树

## 预备知识

基本概念：父亲、祖父、子孙、兄弟。

路径的长即路径上边的个数。

深度为根节点到该节点的路径的长度；高度为节点到叶子节点最长路径的长度。

树的实现

Element、firstChild、nextSibling

树的遍历及应用

## 表达式树

二叉树节点上加入运算符和运算数

先序表达式对应树的先序遍历；

中序表达式对应树的中序遍历；

后序表达式对应树的后序遍历

## 二叉查找树（BST）

Contains、findMin、findMax、insert、remove

## 平衡二叉查找树（AVL）

含义：每个节点的左右子树高度相差不超过1。每次插入、删除操作后，树的结构都会发生变化，在insert、remove操作后，需要进行一次平衡操作，来保证树的平衡性（即树的高度不过深）。树的深度低，其操作的时间复杂度低。所以要保证树平衡性，维持树的性能。

插入会造成树的不平衡，插入完成后，肯定能在插入点到根节点的路径中找到某个点，调整以此点为根的树，来让整个树重新保持平衡。

那么插入有四种情形：左儿子的左子树；左儿子的右子树；右儿子的左子树；右儿子的右子树

这四种情形分别可以通过

## 伸展树（splay tree）

## 再探树的遍历

## B树（balance tree）

首先明确，平衡二叉树和平衡M叉树，查找次数分别为log2N和logMN，M叉树每个节点处要有M-1项，用于生成M项，故M叉树操作次数为(M-1)logMN，求导得知随M增大，这个表达式增大，故数据都在内存中时选择二叉平衡树性能最高。

当数据存储在磁盘时，操作时间=io访问次数\*（io访问时间+cpu在每个节点处的运算时间），io访问时间远大于cpu运算时间，故操作时间= io访问次数\*io访问时间，故尽量减少io次数可以有效提升性能。io访问次数由树的深度决定。树的分叉越多，其深度越低。

给出一些B树的条件：

1. 数据都存储在叶子节点
2. 每个节点的M-1个关键字保证了该节点有M个子树，第i个关键字是第i+1子树的最小值
3. 树的根要么是叶子节点，要么有[2,M]个子树
4. 除根、叶子节点外的节点有M/2到M个子树
5. 叶子节点有L/2到L个数据项（L可以由每个存储块和每条记录的大小计算得出）

新增：插入操作会引起叶子节点超过该叶子节点的最大项数，这时将该叶子节点均分得两个叶子节点，那么其父节点会新增一个关键字，若父节点关键字个数已经最多，那么就分裂这个节点，若分裂这个节点，导致这一层的子树数目超出最大限制，则继续向上分裂。直至根都会产生分裂，产生一个新根，这就是根允许由2个子树的理由。

删除：删除一项后，若该叶子节点中的项数比最低项数还少，那么从相邻节点中领养。若相邻节点中已经是最小项数了，那么将两个叶子节点合并。这可能会导致子树数目小于最小数目，这时需要从相邻处领养，领养不了合并，依次类推，最终会导致根只有一个儿子，此时删除根，让这个儿子作为新的根

# 散列

# 优先队列

Java的PriorityQueue，可用来求解一组数中的topK问题，比如优先队列中先放入3个元素，遍历其他元素，如果该元素大于堆顶元素，则把堆顶元素剔出去，最后遍历完，堆中的元素就是topK。