第一部分

1. 算法的基本特征

- 算法应具备的五个基本特征为：
  ①有穷性：必须在执行有穷步后终止
  ②确定性：算法的每一步均有明确的意义
  ③输入:　0或多个
  ④输出：　1个或多个
  ⑤可行性：每一步的执行时间都是有限的

2. 算法的时间复杂度分析

Σ（指令的一次执行时间*一条指令执行次数）

- 最坏情况运行时间：对规模为n的任何输入，算法的最长运行时间。
  - 一个算法的最坏情况运行时间给出了任何输入的运行时间的一个上界。
  - 对某些算法，最坏算法经常出现。
  - “平均情况”往往与最坏情况大致一样

- 平均情况：概率分析

3. 时间复杂度增长量级

- 增长量级(order of growth)

一个算法的时间是由其增长速率所决定的，所以对于算法时间复杂度的研究主要侧重的是算法的渐进性能，即当问题的输入规模n很大时算法的运行效率。对于一个多项式时间的算法，如：$T(n) = An^2+Bn+C$,当n很大时，T(n)的增长速率是由最高项n^2所决定。

4. 分治算法（递归）

（1）适用问题

- 当求解的问题较复杂或规模较大时，不能立刻得到原问题的解，但这些问题本身具有这样的特点，它可以分解为若干个与原问题性质相类似的子问题，而这些子问题较简单可方便得到它们的解，因此通过合并这些子问题的解就可得到原问题的解。
- 许多有用的算法在结构上是递归的

- 分治法适用条件
- ①原问题可以分解为若干个与原问题性质相类似的子问题
- ②问题的规模缩小到一定程度后可方便求出解
- ③子问题的解可以合并得到原问题的解
- ④分解出的各个子问题应相互独立，即不包含重叠子问题

（2）时间函数

$$T(n)=\begin{cases}\Theta(1) & n \le C \\ aT(\frac{n}{b})+D(n)+C(n) & n > C\end{cases}$$

其中：$\Theta(1)$表示常数时间

a为分解后的子问题个数

$1/b$为分解后子问题与原问题相比的规模

$D(n)$表示分解子问题所花费的时间

$C(n)$表示合并子问题解所花费的时间

5．函数的渐近记号（☆）

（1）θ记号

•def: $\Theta(g(n))=\{f(n)$:存在大于0的常数C_1、C_2和n_0，使得对于所有的$n\ge n_0$，都有$0\le C_1g(n)\le f(n)\le C_2g(n)\}$

•记为：$f(n)=\Theta(g(n))$。

•注意点:

•① 常数C_1、$C_2>0$

•② 只需要存在某个C_1、$C_2>0$即可，不要求对任意的C_1、C_2。

（2）大O记号（渐近上界）

def: $O(g(n))=\{f(n)$:存在大于0的常数C、n_0，使得对于所有的$n\ge n_0$时，都有$0\le f(n)\le Cg(n)\}$

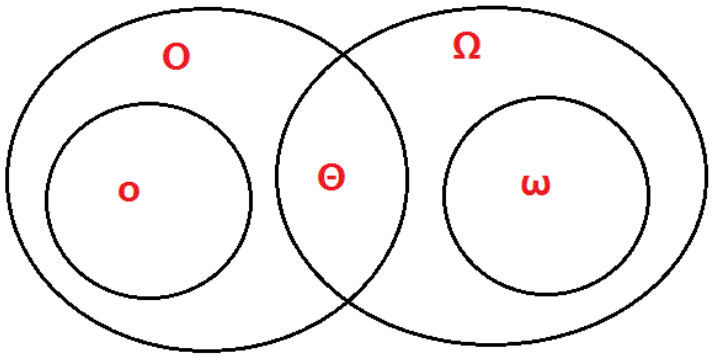
记为：$f(n)=O(g(n))$

（3）大Ω记号（渐近下界）

def: $\Omega(g(n))=\{f(n)$:存在正常数C和n_0，使得对于所有 的$n\ge n_0$，都有$0\le Cg(n)\le f(n)\}$

记为：$f(n)=\Omega(g(n))$

（4）小o，小ω记号（见书）



从上面的图可以看出：
(1)如果 f(n)=Θ(g(n)),则 f(n)=O(g(n))且 f(n)=Ω(g(n))。
(2)如果 f(n)= o (g(n)),则 f(n)=O(g(n))。
(3)如果 f(n)=ω(g(n))，则 f(n)=Ω(g(n))。
(4)如果 f(n)=O(g(n)),则要么是 f(n)= o (g(n))，要么是 f(n)=Θ(g(n))。
(5)如果 f(n)=Ω(g(n)) ,则要么是 f(n)=ω(g(n))，要么是 f(n)=Θ(g(n))。

- 例题

判断题：

$a)f(n)+g(n)=\Theta(\min(f(n),g(n)))$

$b)f(n)=O(g(n))$蕴含$\lg(f(n))=O(\lg(g(n)))$

$c)f(n)=o(g(n))$蕴含$\lg(f(n))=o(\lg(g(n)))$

$d)f(n)=O(g(n))$蕴含$2^{f(n)}=O(2^{g(n)})$

$e)f(n)=O(f^2(n))$

解答：

$a)$错，比如$f(n)=n,g(n)=1$,则$\min(f(n),g(n))=1$,而$n+1\neq\Theta(1)$

$b)$对。

$c)$错，比如$f(n)=n,g(n)=n^2$,但是$\lg(n)\neq o(\lg(n^2))=o(2\lg n)$

$d)$错。$f(n)=n,g(n)=\frac{n}{2}$,但是$2^n\neq O(2^{\frac{n}{2}})$

$e)$错。$f(n)=\frac{1}{n},\frac{1}{n}\neq O(\frac{1}{n^2})$

6.渐近记号的性质
（1）定理 3.1
（2）用于渐近比较

- 自反性

$$\begin{aligned} f(n) &= \Theta(f(n)), \\ f(n) &= O(f(n)), \\ f(n) &= \Omega(f(n)). \end{aligned}$$

- 传递性

$$\begin{aligned} &f(n)=\Theta(g(n)) \text{ and } g(n)=\Theta(h(n)) \text{ imply } f(n)=\Theta(h(n)), \\ &f(n)=O(g(n)) \text{ and } g(n)=O(h(n)) \text{ imply } f(n)=O(h(n)), \\ &f(n)=\Omega(g(n)) \text{ and } g(n)=\Omega(h(n)) \text{ imply } f(n)=\Omega(h(n)), \\ &f(n)=o(g(n)) \text{ and } g(n)=o(h(n)) \text{ imply } f(n)=o(h(n)), \\ &f(n)=\omega(g(n)) \text{ and } g(n)=\omega(h(n)) \text{ imply } f(n)=\omega(h(n)). \end{aligned}$$

- 对称性

$$f(n)=\Theta(g(n)) \text{ if and only if } g(n)=\Theta(f(n)).$$

- 置换性

$$f(n) = O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \quad \text{if and only if} \quad g(n) = \omega(f(n)).$$

7. 递归式求解

（1）代入法

（2）递归树

（3）主方法

8. 概率分析与随机算法

- 概率分析的含义
  - 概率分析的一般思路是：首先对输入的分布情况进行某种假设，最常见的假设当然是均匀分布；然后以此假设为前提，对一个已知的算法进行此种分布下的复杂性和效率分析。
- 随机算法
  - 通过使算法中某部分行为随机化。
  - 一个算法的行为不仅由输入决定，而且也由随机数生成器产生的数值决定，这个算法就是随机的。
  - 在实践中，大多数编程环境提供了一个伪随机数生成器，它是一个确定性算法，返回值在统计上看起来是随机的。
- 随机算法与概率分析的区别

  1）当算法本身做出随机选择时，称之为随机算法。将一个随机算法的运行时间称为期望运行时间。

  2）进行概率分析，必须使用或者假设关于输入的分布，然后分析该算法，计算出一个平均情形下的运行时间。此时称为平均情况运行时间。

第二部分

1. 影响排序时间复杂度的因素

- 时间复杂度依赖于：
  - 将被排序的项数、项数以被排序的程度、项值的可能限制、计算机的体系结构，以及将使用的存储设备的种类。

2. 堆的概念与性质

- 最大堆：对于任意一个拥有父节点的子节点，其数值均不大于父节点的值；这样层层递推，就是根节点的值是最大的。
  - $A[\text{PARENT}(i)] \geq A[i]$
- 最小堆：对于任意一个拥有
  在此处键入公式。父节点的子节点，其数值均不小于父节点的值；这样层层递推，就是根节点的值是最小的。
  - $A[\text{PARENT}(i)] \leq A[i]$

3. 关于比较排序算法的结论

定理 8.1　在最坏情况下，任何比较排序算法都需要做 $\Omega(n \lg n)$ 次比较。

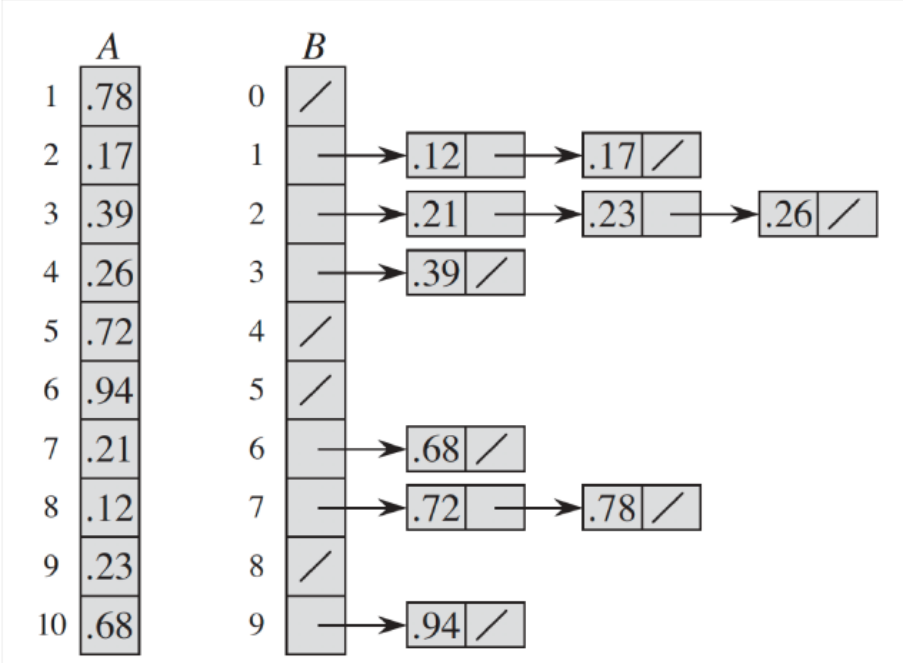
推论 8.2　堆排序和归并排序都是渐近最优的比较排序算法。

4. 计数排序

- 假设 n 个输入元素中的每一个都是介于 0 到 k 之间的整数，此处 k 为某个整数。
- 计数排序的基本思想就是对每一个输入元素x，确定小于x的元素的个数，这样就可以把x直接放在它在最终输出数组的位置上、
  - 例如：有 10 个年龄不同的人，统计出有 8 个人的年龄比 A 小，那 A 的年龄就排在第 9 位，用这个方法可以得到其他每个人的位置，也就排好了序

5. 桶排序

- 桶排序（Bucket Sort）的思想是将数组分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法）。
- 当要被排序的数组内的数值是均匀分配的时候，桶排序可以以线性时间运行。



第三部分

一、二叉搜索树

1. 性质

- 这样的一棵树可以使用一个链表数据结构来表示，其中的每一个节点是一个对象。除了key和卫星数据之外，每个节点还包含属性left（左孩子）、right（右孩子）、和p（双亲）（若不存在，则值为NIL）。
- 设x为二叉树中的任一结点，那么对于x左子树中的任一结点y都有key[y]≤key[x],x右子树中的任一结点y都有key[x]≤key[y]。

- 采用**中序遍历**一棵二叉查找树，可以得到树中关键字由**小到大**的序列。

中序遍历代码：

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

定理12.1：如果x是一棵有n个结点子树的根，那么调用Inorder-tree-walk(x)，需要$\theta(n)$时间。 (证明见 PPT)

2. 二叉搜索树操作

涉及到的操作有 SEARCH, MINIMUN, MAXIMUM, SUCCESSOR, PREDECESSOR

（1） 查找

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

（递归版本）

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

（循环版本）

（2）最小元素、最大元素

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

（3）后继与先驱

1）后继

a）如果一个节点有右子树，它**后继**即是它的右子树的最“左”端的节点，也即右子树的最左节点，

b) 否则它的后继位于它的某个父或祖父点。简单的从x开始沿树而上，直到遇到这样一个结点：这个结点是它双亲的左孩子

2）前驱

a）而一个节点如果有左子树，它的**前驱**即是它的左子树的最“右”端的节点，也即左子树的最大节点，

b）否则它的前驱位于它的某个父或者祖父节点。简单的从x开始沿树而上，直到遇到这样一个结点：这个结点是它双亲的右孩子。

（注：返回的都是遇到的结点的双亲结点，遇到这样的结点只是循环终止条件。）

（4）以上操作均可以在 O（h）时间内完成，h 为树的高度。

3. 二叉搜索树的插入与删除

（1） 插入

- 算法思想

插入结点的位置对应着查找过程中**查找不成功时候的结点位置**，因此需要从根结点开始查找带插入结点位置，找到位置后插入即可。下图所示插入结点过程

- 伪代码

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

- 运行时间为 O（h）

（2）删除

- 算法思想

2 **删除** 从二叉查找树中删除给定的结点z，分三种情况讨论：

① 被删除节点没有孩子。只需要修改其父节点，用NIL去**替换**自己。

② 被删除节点有一个孩子。也只需要修改其父节点，用这个孩子去**替换**自己。

③ 被删除节点有两个孩子。那么先找z的后继y（一定在z的右子树中），并让y占据树中z的位置。z的原来的右子树部分称为y的新的右子树，并且z的左子树成为y的左子树。

a) 如果z的后继y就是z的右孩子（即y没有左孩子），直接用y**代替**z，并保留y的右子树，如右上图所示：

b) 如果z的后继y不是z的右孩子，先用y的右孩子**替换**y，再用y**替换**z。如下图所示：

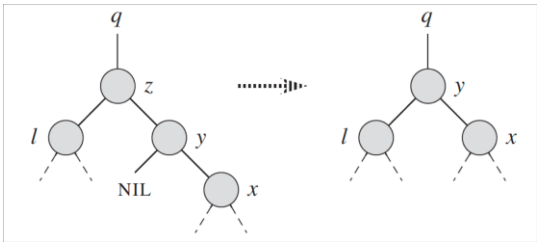
（注：本课程当出现情况 3 时，使用后继，即右子树最左来替换，在数据结构课程中，既可使用后继，也可使用前驱来替换。）

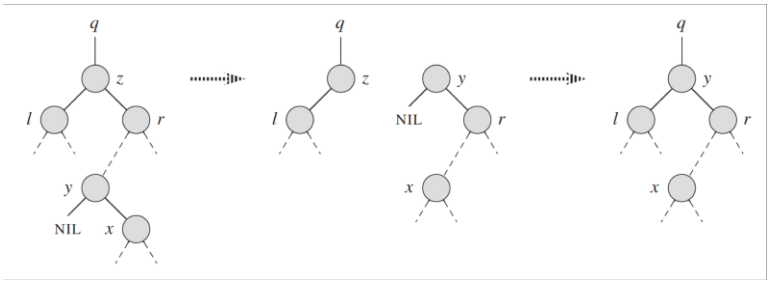
- 伪代码

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)
11     y.left = z.left
12     y.left.p = y
```

- 算法图示





- 删除操作时间复杂度为 O(h)

4. 随机构建二叉搜索树

我们先给出随机构建二叉搜索树的定义：

1) 向一棵空树随机的插入n个关键字而得到的树。

（这里随机的意思是n个关键字的n!种排列都等可能的出现。）

2) 一棵有n个不同关键字的随机构建二叉搜索树的期望高度为O(lgn)　证明

二、红黑树

1. 为什么要使用红黑树？

二叉搜索树的缺陷：失去平衡

“**平衡**”的大致意义是：没有任何一个节点过深（深度过大）：不同的平衡条件，造就出不同的效率表现，以及不同的实现复杂度。

红黑树可实现出平衡二叉搜索树：

确保没有一条路径会比其他路径长出俩倍，因而是接近平衡的。

2. 红黑树的性质

（1）二叉搜索树的性质红黑树都具有

（2） 自身五个特性

- ①每个结点要么是黑色要么是红色
- ②树根结点的颜色为黑色
- ③叶结点(nil)为黑色
- ④如果某个结点为红色，则它的左、右孩子结点均为黑色
- ⑤对树中任一结点，所有从该结点出发到其叶结点的路径中均包含相同数目的黑色结点

（3） 每个结点有 p，color，left，right，key

3. 黑高

（1）定义

- 黑高bh(x)表示从x结点出发(不包含x结点)到其叶结点路径上的黑色结点个数。

（2） 黑高的性质

1) 根据红黑树的"**特性(5)**，即*从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点*"可知，从节点x出发到达的所有的叶节点具有相同数目的黑节点。**这也就意味着，bh(x)的值是唯一的**！

2) 根据红黑色的"**特性(4)**，即*如果一个节点是红色的，则它的子节点必须是黑色的*"可知，从节点x出发达到叶节点"所经历的黑节点数目">= "所经历的红节点的数目"。假设x是根节点，则可以得出结论"**bh(x) >= h/2**"。

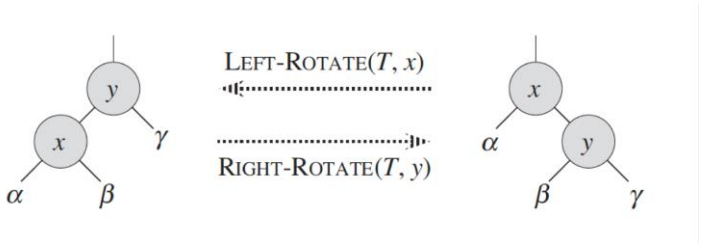
- 具有n个内部结点的红黑树高度最多为2lg(n+1)。

4. 红黑树操作

基本操作时添加、删除；会用到变色和旋转方法。

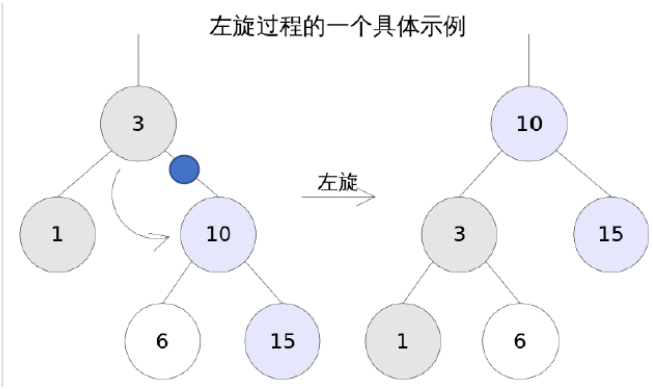
（1） 变色

（2） 旋转



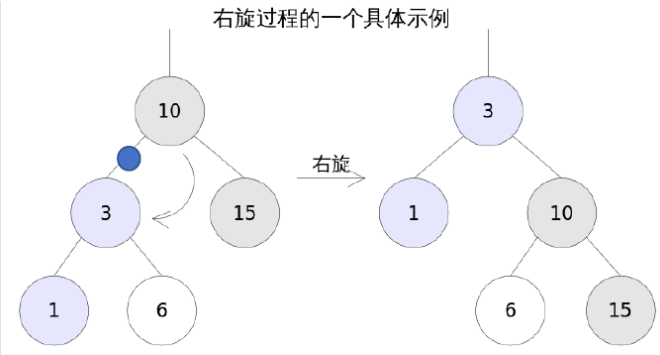
- 左旋

1）对x进行**左旋**，意味着，将“x的右孩子”设为“x的父亲节点”；即，将x变成了一个左节点(x成了为z的左孩子)！。因此，**左旋中的“左”，意味着“被旋转的节点将变成一个左节点”**。



- 右旋

2）对x进行**右旋**，意味着，将“x的左孩子”设为“x的父亲节点”；即，将x变成了一个右节点(x成了为y的右孩子)！因此，**右旋中的“右”，意味着“被旋转的节点将变成一个右节点”**。



（3） 插入操作

算法的时间复杂度为0（lgn）

插入一个新结点分为二步完成：

1. 按照二叉查找树的方式插入新结点z
2. **将插入的节点着色为“红色”。**

（不会违背"特性(5)"！）

3. 恢复红黑树的特性

难点：恢复红黑树特性，分为三种情况

（4）删除操作

算法的时间复杂度为0（lgn）

难点，待补充

三、散列表 开放寻址法

1. 开放寻址法（具体可以参考数据结构，注：二次探查有些不同）

- 在开放寻址法（open addressing）中，所有的元素存放在散列表里。也就是说，每个表项或包含动态集合的一个元素，或包含NIL。
- 当查找某个元素时，要系统地检查所有的表项，直到找到所需的元素，或者最终查明该元素不在表中。
- 该方法导致的一个结果就是转载因子alpha不会超过1。

2. 线性探查

$$h(k, i) = (h'(k) + i) \bmod m,\ i = 0, 1, \ldots, m-1$$

3. 二次探查

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m\ ,$$

4. 双重散列

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$$

第四部分

一、动态规划

1. 动态规划概述

- 动态规划（dynamic programming）。
  - 主要用于优化问题求解，即求出问题的最优(最大/小)解，当有多个最优解时一般是求一个即可。
    - Note：优化问题通常有很多可行解，每个解法都有一个值，希望找到具有最优值（最小值或者最大值）的解。这样的解为一个最优解，有可能会有多个解都能得到最优值。
  - 应用于求解子问题重叠的情况，也就是不同的子问题会涉及相同的子子问题。
    - 动态规划则是对公共子问题只求解一次，然后将其解保存在表格中，避免了不必要的重复工作。

2. 与分治法比较

- 动态规划与分治法异同
  - 相同点：
    - 都是通过组合子问题的解来求解原问题。
  - 不同点：
    - 1)分治法是将问题划分为独立的子问题，递归地求解子问题，然后combine
    - 2)当分解问题非独立时，即他们共享子子问题时，可采用动态规划。

- 分治法与动态规划的比较
  - 当递归每一步产生一个新的子问题时，适合使用分治法
  - 当递归中较多出现重叠子问题时，适合使用动态规划，即对重叠子问题只求解一次，然后存储在表中，当需要使用时常数时间内查表。若子问题规模是多项式阶的，动态规划特别有效。

3. 动态规划的两种实现方法

（1）带备忘的自顶向下法：在递归算法调用每一层的时候，先检查该值有没有被计算过，若没有，调用并存储，若计算过，直接取出该值。

（2）自底向上法：将一个问题分成规模更小的子问题，从小到大进行求解，当求解至原问题时，所需的值都已求解完毕。

4. 动态规划的适用问题

- 什么样的优化问题适合使用动态规划?
  - 最优子结构　　分治，DP，Greedy
  - **重叠子问题**　　DP is the best！

5. 动态规划问题的要素

- 最优子结构
- 细节
- 重叠子问题
- 重构最优解
- 备忘

6. 相关问题

钢条分割、矩阵链乘、最长公共子序列、最优二叉搜索树

二、贪心算法

1. 贪心算法概述

- **狭义**的贪心算法：解最优化问题的一种特殊方法。
  - 解决过程中总是做出当下最好的选择，因为具有最优子结构的特点，局部最优解可以得到全局最优解；
  - 这种贪心算法是动态规划的一种特例。**能用贪心解决的问题，也可以用动态规划解决。**
- **广义**的贪心算法：一种通用的贪心策略，基于当前局面而进行贪心决策。
  - 以**跳一跳**的题目为例：
  - 我们发现的题目的核心在于**向右能到达的最远距离**，我们用maxRight来表示；
  - 此时有一种贪心的策略：从第1个盒子开始向右遍历，对于每个经过的盒子，不断更新maxRight的值。

将最优化问题转化为这样一个问题，即先做出选择，再解决剩下的一个子问题。

证明原问题的最优解之一可以由贪心选择得到。

证明在做出贪心选择后，将剩余的子问题的最优解和我们所做的贪心选择组合起来，可以得到原问题的一个最优解。

- 贪心算法是通过做一系列选择来获得最优解，在算法里的每一个决策点上，都力图选择最好的方案，这种启发式策略并非总能产生最优解。

2. 贪心算法与动态规划的比较

- 相同点：两种方法都利用了最优子结构特征
- 易错误处：
  - ① 当贪心算法满足全局最优时，可能我们试图使用动态规划求解，但前者更有效
  - ② 当事实上只有动态规划才能求解时，错误地使用了贪心法

3. 应用

- Prim 算法
- 迪杰斯特卡算法
- 构造哈夫曼树
- 背包问题

三、摊还分析

1. 摊还分析概念

求数据结构的一个操作序列中所执行的所有操作的平均时间，来评价操作的代价

2. 三种方法

①聚合分析：这种方法用来确定一个n个操作序列的总代价的上界T(n)，因此每个操作的平均代价为T(n)/n

②核算法：用来分析**每个操作的摊还代价**。核算法将序列中某些**较早的操作的“余额”作为“预付信用”**存储起来，与数据结构中的特定对象相关联。在操作序列中随后的部分，存储的信用即可用来为拿些缴费少于实际代价的操作支付差额

③势能法：也是分析**每个操作的摊还代价**，也是通过较早的操作的余额来补偿稍后操作的差额。势能法**将信用作为数据结构的势能存储起来**，且将势能作为一个整体存储，而不是将信用与数据结构中单个对象关联分开存储。

3. 二进制计数器递增问题

- **二进制计数器递增**
  - k位二进制计数器递增的问题
  - 计数器的初值为0.
  - 用一个位数组A[0..k-1]作为计数器，其中A.length = k。
  - 当计数器中保存的二进制值为x时，x的最低位保存在A[0]，而最高位保存在A[k-1]中
    因此$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$
- 操作：加1
- 算法：Increment(A)，初始时x=0，因此对所有$i = 0,1,\ldots,k-1, A[i] = 0$。

4. 核算法

- 核算法思想：先对每个不同的操作核定一个不同的费用可以大于或者小于实际的代价，但不能为负值），称其为**摊还代价**；然后计算n次操作总的费用。
- 这种事先的操作代价核定可能与实际代价不符，存在二种情况：

1）超额收费：摊还代价>实际代价，此时差额（称其为**信用/存款**。）存放在该对象的身上，

2）收费不足：摊还代价<实际代价，此时差额由该对象身上的存款支付

- 摊还分析是否正确：需检查n次操作核定**摊还总代价**是否大于n次**操作实际总费用**

若令C_i为第i次操作的实际费用

$\hat{C}_i$为第i次操作的摊还代价

那么要求：

$$\sum_{i=1}^{n}\hat{C}_i \geq \sum_{i=1}^{n}C_i \quad \text{(对任意的n)}$$

$$\text{信用}=\sum_{i=1}^{n}\hat{C}_i-\sum_{i=1}^{n}C_i$$

5．使用核算法分析二进制计数递增

二进制计数器递增摊还代价核定：

- ∵每个操作的实际成本为0-1翻转的位数，令**翻转1次的实际费用为1元**
  1）置位操作(0→1)：核定2元，**1元用于支付实际成本**，另一元作为存款存放在刚完成置位的1身上
  2）复位操作(1→0)：核定0元，**1元的实际成本**由其身上的存款支付

- 摊还代价核定验证：

∵计数器初态为0

∴**置位先于复位**

∵置位时2元中1元支付置位时的实际成本，另1元作为存款存放在该置位的1身上

∴任何时刻计数器**每位1身上均有1元存款**可支付复位时的实际成本

$\therefore \sum_{i=1}^{n}\hat{C}_i-\sum_{i=1}^{n}C_i \geq 0$

∵最坏情况下n个操作均为置位操作且总置位次数≤n

∴总成本T(n)≤2n=O(n)

- 每个操作的平摊时间为O(1)。

6．势能法

1）令n个操作的数据结构为D，数据结构的初态为D_0

2）C_i：表示第i次操作的实际代价

3）C_i^：表示第i次操作的摊还代价

4）D_i：表示在数据结构状态为D_{i-1}上完成第i次操作后的数据结构状态，

即：　　D_{i-1}====>D_i，i=1,2,…,n

5）Φ：势函数

Φ(D_i)：表示将Di映射为一实数，这个实数值称为势能

$$C_i\hat{}=C_i+\Phi_i-\Phi_{i-1}$$

7. 使用势能法分析二进制计数递增

（1）由于**计数器初始状态为0**，而1的个数不会为负数，所以n个操作的总摊还代价为总实际代价的上界O(n)。

（2）若**计数器初态不为0时**

令初始势能$\Phi_0 = b_0 > 0$，　$\Phi_n = b_n$

$\because 0 \le b_0$，　$b_n \le k$

$$\therefore \sum_{i=1}^{n} C_i = \sum_{i=1}^{n} \hat{C}_i - \Phi_n + \Phi_0 \qquad b_0 \le k$$

$$\le \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0 \le 2n + b_0$$

$$\le 2n + k \le 3n = O(n) \quad 当 k = O(n)$$

所以总摊还代价的上界为O(n)