2024Fall 算法基础徐班内容 复习自用

Ch2

插入排序

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

循环不变式三条性质：

初始化：循环的第一次迭代之前，它为真。

保持：如果循环的某次迭代之前它为真，那么下次迭代之前它仍为真。

终止：在循环终止时仍为真，不变式为我们提供一个有用的性质，该性质有助于证明算法是正确的。

分治思想：

这些算法典型地遵循分治法的思想：将原问题分解为几个规模较小但类似于原问题的子问题，递归地求解这些子问题，然后再合并这些子问题的解 来建立原问题的解。 分治模式在每层递归时都有三个步骤：分解原问题为若干子问题，这些子问题是原问题的规模较小的实例。 解决这些子问题，递归地求解各子问题。然而，若子问题的规模足够小，则直接求解。 合并这些子问题的解成原问题的解。

> 分治算法运行时间的递归式来自基本模式的三个步骤。如前所述，我们假设 $T(n)$是规模为 n 的一个问题的运行时间。若问题规模足够小，如对某个常量 c，$n \leqslant c$，则直接求解需要常量时间，我们将其写作 $\Theta(1)$。假设把原问题分解成 a 个子问题，每个子问题的规模是原问题的 $1/b$。（对归并排序，a 和 b 都为 2，然而，我们将看到在许多分治算法中，$a \neq b$。）为了求解一个规模为 n/b 的子问题，需要 $T(n/b)$的时间，所以需要 $aT(n/b)$的时间来求解 a 个子问题。如果分解问题成子问题需要时间 $D(n)$，合并子问题的解成原问题的解需要时间 $C(n)$，那么得到递归式：
>
> $$T(n) = \begin{cases} \Theta(1) & \text{若 } n \leqslant c \\ aT(n/b) + D(n) + C(n) & \text{其他} \end{cases}$$

归并排序

```
MERGE(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1..n1+1] and R[1..n2+1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    L[n1+1] = inf, R[n2+1] = inf
    i = 1, j = 1
```

```
    for k = p to r
        if L[i] <= R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1

MERGESORT(A, p, r)
    if p < r
        q = (p + r)/2
        MERGESORT(A, p, q)
        MERGESORT(A, p+1, r)
        MERGE(A, p, q, r)
```

习题：

改写归并排序可以使计算逆序对的时间复杂度降低到 $nlogn$；

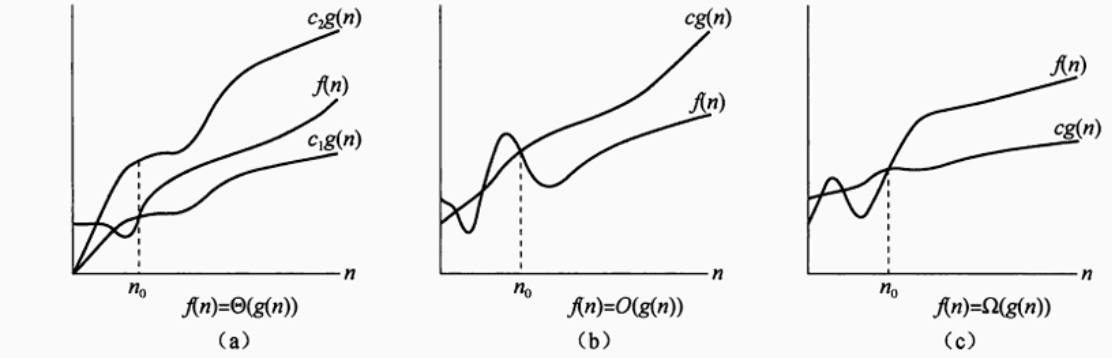
虽然归并排序的最坏情况运行时间为 (nlgn), 而插入排序的最坏情况运行时间为 n^2, 但是插入排序中的常量因子可能使得它在n较小 时，在许多机器上实际运行得更快。因此，在归并排序中当子问题变得足够小时，采用插入排序来使递归的**叶变粗、树变矮**是有意义的。

Ch3

不同的渐进记号：

记号	含义	通俗理解
(1)Θ（西塔）	紧确界。	相当于"="
(2)O （大欧）	上界。	相当于"<="
(3)o（小欧）	非紧的上界。	相当于"<"
(4)Ω（大欧米伽）	下界。	相当于">="
(5)ω（小欧米伽）	非紧的下界。	相当于">"

图 3-1(a)给出了函数 $f(n)$与 $g(n)$的一幅直观画面，其中 $f(n)=\Theta(g(n))$。对在 n_0 及其右边 n 的所有值，$f(n)$的值位于或高于 $c_1g(n)$且位于或低于 $c_2g(n)$。换句话说，对所有 $n\geqslant n_0$，函数 $f(n)$ 在一个常量因子内等于 $g(n)$。我们称 $g(n)$是 $f(n)$的一个**渐近紧确界**(asymptotically tight bound)。



Ch4

本章介绍三种求解递归式的方法，即得出算法的“Θ”或“O”渐近界的方法：

- **代入法**　我们猜测一个界，然后用数学归纳法证明这个界是正确的。
- **递归树法**　将递归式转换为一棵树，其结点表示不同层次的递归调用产生的代价。然后采用边界和技术来求解递归式。
- **主方法**　可求解形如下面公式的递归式的界：

$$T(n) = aT(n/b) + f(n) \tag{4.2}$$

其中 $a \geqslant 1$，$b > 1$，$f(n)$是一个给定的函数。这种形式的递归式很常见，它刻画了这样一个分治算法：生成 a 个子问题，每个子问题的规模是原问题规模的 $1/b$，分解和合并步骤总共花费时间为 $f(n)$。

最大子数组的分治求解：

最大子数组的三种情况：1 在左半边，2 在右半边，3 横跨（从 mid 向两边扩展即可，线性的）

运行时间：$T(n) = 2T(n/2) + \Theta(n)$

最大子数组的线性求解：（dp）

```
/* 在数组a[l,r]内找到最大子数组（非空!） */
SUB_ARRAY FindMaxSubarray1(int a[], int l, int r) {
    int ans = INT_MIN, cur = INT_MIN;
    int curLeft, leftIndex, rightIndex;
    //记录cur对应的子数组左下标、记录ans对应的子数组左右下标
    for (int i = l; i <= r; i++) {
        /* 找到a[ 0..i ]内包含第i项的最大子数组：要么增加a[i]，要么就是a[i] */
        if (cur + a[i] >= a[i])
            cur += a[i];
        else {
            cur = a[i];
            curLeft = i;
        }
        /* 更新a[ 0..i ]内的最大子数组 */
        if (cur > ans) {
            ans = cur;
            leftIndex = curLeft;
            rightIndex = i;
        }
    }
}
```

矩阵乘法的 Strassen 算法：

$$T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{若 } n > 1 \end{cases} \tag{4.18}$$

我们用常数次矩阵乘法的代价减少了一次矩阵乘法。一旦我们理解了递归式及其解，就会看到这种交换确实能带来更低的渐近运行时间。利用 4.5 节的主方法，可以求出递归式(4.18)的解为 $T(n) = \Theta(n^{\lg 7})$。

代入法解递归式：

做做题。主打一个猜测和经验

递归树解递归式：

每层的代价（结点数*单层代价）求和，或者猜

主方法解递归式：

主定理的标准形式是分析以下递归式的实际复杂度：

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

其中：

- $a \geq 1$ 是递归调用的数量，表明问题被切割为几个子问题。
- $b > 1$ 是每次递归将问题规模缩小的因子，表明问题子问题的规模。
- $f(n)$ 是所有递归调用以外的额外工作量。

主定理的核心思想是比较递归的部分和非递归部分哪个增长得更快，从而决定算法的总体复杂度。根据 $f(n)$ 的增长速度与 $n^{\log_b a}$ 的比较，主定理分为以下三种情形：

1. $f(n) \in O(n^{\log_b a - \epsilon})$，**某个** $\epsilon > 0$
   此时递归部分主导增长，时间复杂度为 $T(n) \in \Theta(n^{\log_b a})$。

2. $f(n) \in \Theta(n^{\log_b a})$
   此时递归部分与非递归部分增长相当，时间复杂度为 $T(n) \in \Theta(n^{\log_b a} \log n)$。

3. $f(n) \in \Omega(n^{\log_b a + \epsilon})$，**某个** $\epsilon > 0$ **且** $af\left(\frac{n}{b}\right) \leq cf(n)$ **（即递归调用的贡献足够小）**
   此时非递归部分主导增长，时间复杂度为 $T(n) \in \Theta(f(n))$。

简单的理解：

1. 若 f(n) 增长较慢（情况 1），则递归部分占主导。
2. 若 f(n) 和递归增长相等（情况 2），需额外乘以 logn。
3. 若 f(n) 增长较快且满足平衡条件（情况 3 -- c < 1），则非递归部分占主导。

Ch6

堆排序

优点：时间复杂度是O(nlgn)；具有空间原址性：任何时候都只需要常数个额外的元素空间存储临时数据。堆排序集合了归并排序和快速排序的优点。

最大堆性质：除了根节点之外的所有节点 i 都满足 A[parent(i)] >= A[i] （大根堆）（最小堆亦然）

```
MAX-HEAPIFY(A, i) // 维护大根堆性质 // 可以改写为非递归
    l = i.left
    r = i.right
    if l <= A.heap_size and A[l] > A[i] largest = l
    else if r <= A.heap_size and A[r] > A[i] largest = r
    else largest = i
    if largest != i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest) // 向子树递归
```

建堆：自底向上

```
BUILD_MAX_HEAP(A)
    A.heap_size = A.length
    for i = A.length / 2 downto 1
        MAX_HEAPIFY(A, i)
```

为了证明 BUILD-MAX-HEAP 的正确性，我们使用如下的循环不变量：

在第 2～3 行中每一次 **for** 循环的开始，结点 $i+1$，$i+2$，…，n 都是一个最大堆的根结点。

我们需要证明这一不变量在第一次循环前为真，并且每次循环迭代都维持不变。当循环结束时，这一不变量可以用于证明正确性。

初始化：在第一次循环迭代之前，$i=\lfloor n/2 \rfloor$，而$\lfloor n/2 \rfloor+1$，$\lfloor n/2 \rfloor+2$，…，n 都是叶结点，因而是平凡最大堆的根结点。

保持：为了看到每次迭代都维护这个循环不变量，注意到结点 i 的孩子结点的下标均比 i 大。所以根据循环不变量，它们都是最大堆的根。这也是调用 MAX-HEAPIFY(A, i)使结点 i 成为一个最大堆的根的先决条件。而且，MAX-HEAPIFY 维护了结点 $i+1$，$i+2$，…，n 都是一个最大堆的根结点的性质。在 **for** 循环中递减 i 的值，为下一次循环重新建立循环不变量。

终止：过程终止时，$i=0$。根据循环不变量，每个结点 1，2，…，n 都是一个最大堆的根。特别需要指出的是，结点 1 就是最大的那个堆的根结点。

堆排序算法：从叶子到根

```
HEAPSORT(A)
  for i = A.length downto 2
    BUILD_MAX_HEAP(A)
    exchange A[1] with A[i]
    A.heap_size--
    MAX_HEAPIFY(A, 1)
```

HEAPSORT 过程的时间复杂度是 $O(n\lg n)$，因为每次调用 BUILD-MAX-HEAP 的时间复杂度是 $O(n)$，而 $n-1$ 次调用 MAX-HEAPIFY，每次的时间为 $O(\lg n)$。

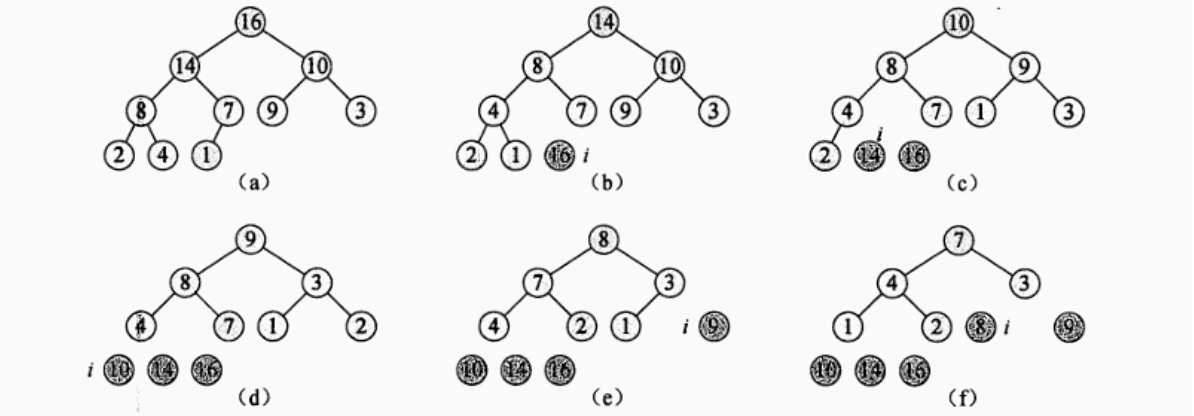


图 6-4 HEAPSORT 的运行过程。(a)执行堆排序算法第 1 行，用 BUILD-MAX-HEAP 构造得到的最大堆。(b)～(j)每次执行算法第 5 行，调用 MAX-HEAPIFY 后得到的最大堆，并标识当次的 i 值。其中，仅仅浅色阴影的结点被保留在堆中。(k)最终数组 A 的排序结果

插入元素：插到最后一片叶子，沿父亲跳到合适的位置

删除元素：找到该元素，将该结点的值设置为根节点的值，将根节点的值设置为最后一片叶子的值（根此时最小），heap_size -1，然后 MAX_HEAPIFY(A, 1) 。

优先队列

用堆来实现就行。

Ch7

快速排序：

原址排序，不稳定；关键步骤是**划分**

分解：数组 $A[p..r]$被划分为两个(可能为空)子数组 $A[p..q-1]$和 $A[q+1..r]$，使得 $A[p..q-1]$中的每一个元素都小于等于 $A[q]$，而 $A[q]$也小于等于 $A[q+1..r]$中的每个元素。其中，计算下标 q 也是划分过程的一部分。

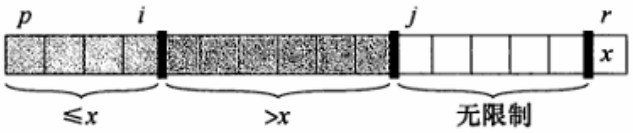
解决：通过递归调用快速排序，对子数组 $A[p..q-1]$和 $A[q+1..r]$进行排序。

合并：因为子数组都是原址排序的，所以不需要合并操作：数组 $A[p..r]$已经有序。

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
    pivot = A[r]
    i = p - 1
    for j = p to r-1
        if A[j] <= pivot
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
```

```
    return i+1
```



在子数组 $A[p..r]$上，PARTITION 维护了 4 个区域。$A[p..i]$区间内的所有值都小于等于 x，$A[i+1..j-1]$区间内的所有值都大于 x，$A[r]=x$。子数组 $A[j..r-1]$中的值可能属于任何一种情况

快速排序的循环不变量是如下一些性质：

对任意数组下标 k 有：

若 p <= k <= i，则 A[k] <= pivot；若 i + 1 <= k <= j - 1，则 A[k] > pivot

若 k = r 则 A[k] = pivot

快排的性质：

最坏时间复杂度 $O(n^2)$ - 最不平衡划分

最好时间复杂度 $O(nlogn)$ - 最好情况划分，均分

> 事实上，任何一种常数比例的划分都会产生深度 为 \Theta{logn} 的递归树，其中每一层的时间代价都是O(n)。因此，只要划分是常数比例的，算法的运行时间总是O(nlgn)

使用 RANDOMIZED-PARTITION, 在输入元素互异的情况下，快速排序算法的期望运行时间为O(nlgn)。

Ch8

线性时间排序：不用比较操作，而用运算来确定顺序

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。

证明 根据前面的讨论，对于一棵每个排列都是一个可达的叶结点的决策树来说，树的高度完全可以被确定。考虑一棵高度为 h、具有 l 个可达叶结点的决策树，它对应一个对 n 个元素所做的比较排序。因为输入数据的 $n!$ 种可能的排列都是叶结点，所以有 $n! \leqslant l$。由于在一棵高为 h 的二叉树中，叶结点的数目不多于 2^h，我们得到：

$$n! \leqslant l \leqslant 2^h$$

对该式两边取对数，有

$$\begin{aligned} h &\geqslant \lg(n!) && \text{（因为 lg 函数是单调递增的）} \\ &= \Omega(n\lg n) && \text{（由公式(3.19)）} \end{aligned}$$

■

> 在一棵 n 个元素比较排序算法的决策树中，一个叶节点可能的最小深度是 n-1。

计数排序：

计数排序假设 n 个输入元素中的每一个都是在 0 到 k 区间内的一个整数，其中k为某个整数。

当$k = O(n)$时，排序的运行时间为 $\Theta(n)$。 计数排序是稳定排序。

计数排序的时间代价是多少呢？第 2～3 行的 **for** 循环所花时间为 $\Theta(k)$，第 4～5 行的 **for** 循环所花时间为 $\Theta(n)$，第 7～8 行的 **for** 循环所花时间为 $\Theta(k)$，第 10～12 行的 **for** 循环所花时间为 $\Theta(n)$。这样，总的时间代价就是 $\Theta(k+n)$。在实际工作中，当 $k=O(n)$时，我们一般会采用计数排序，这时的运行时间为 $\Theta(n)$。

```
//假设输入是一个数组A[1..n], A. length=n。我们还需要两个数组： B[1.. n]存放排序的输出，C[0.. k]提供临时存储空间。

COUNTING_SORT(A, B, k)
    let C[0..k] be a new array
    for i = 0 to k: C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of elements equal to A[j]
    for i = 1 to k
        C[i] = C[i] + C[i-1]
    // C[i] now contains the number of elements less than or equal to A[j]
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1 // 这是为了有相同元素时仍能顺利输出
```

8.2-4 设计一个算法，它能够对于任何给定的介于 0 到 k 之间的 n 个整数先进行预处理，然后在 $O(1)$时间内回答输入的 n 个整数中有多少个落在区间$[a..b]$内。你设计的算法的预处理时间应为 $\Theta(n+k)$。

执行计数排序的前两个循环即可，答案是 $C[b]-C[a-1]$。

基数排序：

基数排序的代码是非常直观的。在下面的代码中，我们假设 n 个 d 位的元素存放在数组 A 中，其中第 1 位是最低位，第 d 位是最高位。

RADIX-SORT(A, d)
1 **for** $i=1$ **to** d
2 　　use a stable sort to sort array A on digit i

引理 8.3 给定 n 个 d 位数，其中每一个数位有 k 个可能的取值。如果 RADIX-SORT 使用的稳定排序方法耗时 $\Theta(n+k)$，那么它就可以在 $\Theta(d(n+k))$时间内将这些数排好序。

证明 基数排序的正确性可以通过对被排序的列进行归纳而加以证明(见练习 8.3-3)。对算法时间代价的分析依赖于所使用的稳定的排序算法。当每位数字都在 0 到 $k-1$ 区间内(这样它就有 k 个可能的取值)，且 k 的值不太大的时候，计数排序是一个好的选择。对 n 个 d 位数来说，每一轮排序耗时 $\Theta(n+k)$。共有 d 轮，因此基数排序的总时间为 $\Theta(d(n+k))$。 ■

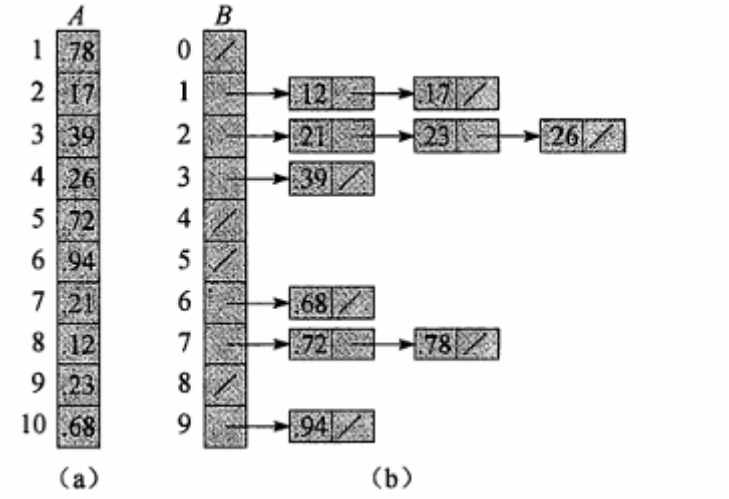
为什么要从低位到高位排：首先是为了保证相同高位的数字可以能够按照低位排序，而且如果高位优先，那么高位排好的序在低位又会被打乱。

基数排序是稳定排序。

```
RADIX_SORT(A, B, d) // 假设数据不超过 d 位
    for i = 0 to d-1
    // use counting-sort to sort array A on digit i
        for j = 1 to A.length
            digit = (A[j] / (10*i)) % 10
            C[digit] = C[digit] + 1
        for each element C[i] of C, C[i] = C[i] + C[i-1]
        for j = A.length downto 1
            B[C[digit]] = A[j] (where j coresponds with digit)
            C[digit]--
```

桶排序：

桶排序将[0，1)区间划分为 n 个相同大小的子区间，或称为**桶**。然后，将 n 个输入数分别放到各个桶中。因为输入数据是均匀、独立地分布在[0，1)区间上，所以一般不会出现很多数落在同一个桶中的情况。为了得到输出结果，我们先对每个桶中的数进行排序，然后遍历每个桶，按照次序把各个桶中的元素列出来即可。



（习题 8.4-2）当所有元素都落到同一个桶中，桶排序退化为快速排序；

优化方案：对每个桶中的元素不用快排而是归并排序，etc.

习题：

8-2、8-5

Ch9

在一个由 n 个元素组成的集合中，**第 i 个顺序统计量**(order statistic)是该集合中第 i 小的元 素。

单独找到最大值或最小值需要 n-1 次比较，同时找到最大和最小值需要 3*[n/2] 次比较。

也可以用决策树解题。

Ch13

红黑树的性质：

- 红黑性质：

1. 每个结点或是红色的，或是黑色的。
2. 根结点是黑色的。
3. 每个叶结点(NIL)是黑色的。
4. 如果一个结点是红色的，则它的两个子结点都是黑色的。
5. 对每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点。

- 其他性质：

任何一条从根到叶子的简单路径，不会超过其他简单路径长度的2倍

从某个结点x出发（**不含该结点**）到达一个叶结点的任意一条简单路径上的黑色结点个数称为该结点的黑高(black-height)，记为 bh(x)。设 h 为树高，则根节点的黑高（也就是红黑树的黑高）至少为 h/2

设内部节点个数为 n，则其与树高 h 有关系式 n >= 2^{h/2} -1

(13.1-6) 在一棵黑高为 K 的红黑树中，内部结点最多可能有多少个？最少可能有多少个？

最多：红黑交替，$2^{2k+1}-1$；最少：全黑，$2^{k+1}-1$

Ch14

顺序统计树（Order-Statistic Tree）：顺带维护了结点 size 属性的红黑树

查找OS树中第 i 小（秩为 i，秩是中序遍历顺序）的元素：

```
OS-SELECT(x, i) // recursive
    r = x.left.size + 1
    if i == r then return x
    else if i < r
        return OS-SELECT(x.left, i)
    else return OS-SELECT(x.right, i - r)


OS-SELECT(x, i) // non-recursive
    r = x.left.size + 1
    while(i != r) do
        if i > r
            x = x.right
            i = i - r
        else
            x = x.left
        r = x.left.size
    return x
```

确定一个元素的秩：

```
OS-RANK(T, x) // non-recursive
    r = x.left.size + 1
```

```
    y = x
    while y != T.root
        if y == y.p.right
            r = r + y.p.left.size + 1 // 如果当前在右子树，由中序遍历，要加上左子树和根
        y = y.p
    return r

OS-RANK(T, x) // recursive
    if(x == T.root)
        return T.root.left.size + 1
    else if(T.root > x)
        return OS-RANK(T.left, x)
    else return OS-RANK(T.right, x) + T.root.left.size + 1
```

(14.1-7) 利用顺序统计树对一个数组中的逆序对计数：

逆序对个数 = \Sum{ 元素在数组中的下标 - 在顺序统计树中的秩 }

定理 14.1(红黑树的扩张) 设 f 是 n 个结点的红黑树 T 扩张的属性，且假设对任一结点 x，f 的值仅依赖于结点 x、$x.left$ 和 $x.right$ 的信息，还可能包括 $x.left.f$ 和 $x.right.f$。那么，我们可以在插入和删除操作期间对 T 的所有结点的 f 值进行维护，并且不影响这两个操作的 $O(\lg n)$ 渐近时间性能。

证明 证明的主要思想是，对树中某结点 x 的 f 属性的变动只会影响到 x 的祖先。也就是说，修改 $x.f$ 只需要更新 $x.p.f$，改变 $x.p.f$ 的值只需要更新 $x.p.p.f$，如此沿树向上。一旦更新到 $T.root.f$，就不再有其他任何结点依赖于新值，于是过程结束。因为红黑树的高度为 $O(\lg n)$，所以改变某结点的 f 属性要耗费 $O(\lg n)$ 时间，来更新被该修改所影响的所有结点。

如果扩张属性只影响到父节点或只影响到子节点，就可以在红黑树上扩张。

区间树（Interval Tree）：

图 14-4 说明了区间树是如何表达一个区间集合的。我们将按照 14.2 节中的 4 步法，来分析区间树以及区间树上各种操作的设计。

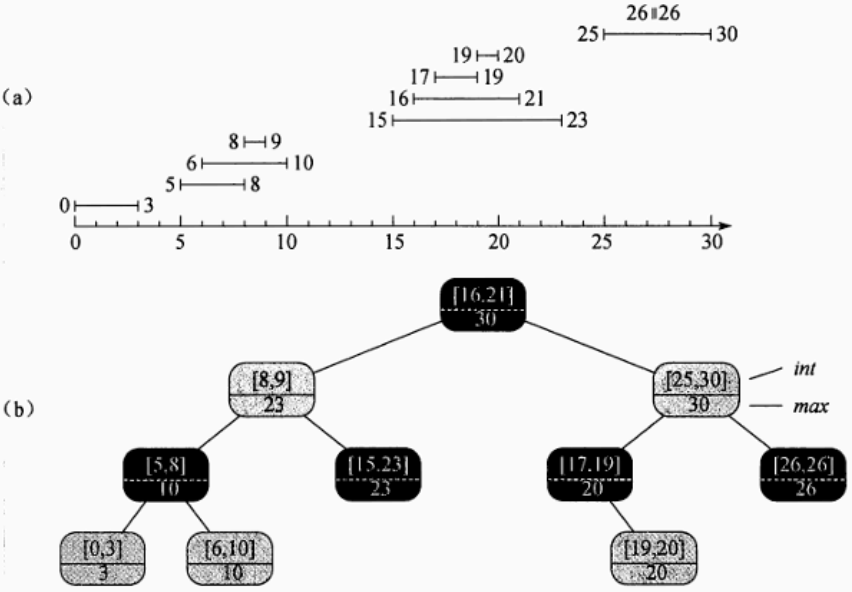


图 14-4 一棵区间树。(a)10 个区间的集合，它们按左端点自底向上顺序示出。(b)表示它们的区间树。每个结点 x 包含一个区间，显示在虚线的上方；一个以 x 为根的子树中所包含的区间端点的最大值，显示在虚线的下方。这棵树的中序遍历列出按左端点顺序排列的各个结点

步骤 1：基础数据结构

我们选择这样一棵红黑树，其每个结点 x 包含一个区间属性 $x.int$，且 x 的关键字为区间的低端点 $x.int.low$。因此，该数据结构按中序遍历列出的就是按低端点的次序排列的各区间。

步骤 2：附加信息

每个结点 x 中除了自身区间信息之外，还包含一个值 $x.max$，它是以 x 为根的子树中所有区间的端点的最大值。

步骤 3：对信息的维护

我们必须验证 n 个结点的区间树上的插入和删除操作能否在 $O(\lg n)$ 时间内完成。通过给定区间 $x.int$ 和结点 x 的子结点的 max 值，可以确定 $x.max$ 值：

$$x.max = \max(x.int.high, x.left.max, x.right.max)$$

这样，根据定理 14.1 可知，插入和删除操作的运行时间为 $O(\lg n)$。事实上，在一次旋转后，更新 max 属性只需 $O(1)$ 的时间，如练习 14.2-3 和练习 14.3-1 所示。

```
INTERVAL-SEARCH(T, i) // 找出所有重叠区间
    x = T.root
    if x == null then return null
    if i overlaps x.interval
        record x.interval
    if x.left != null && x.left.max >= i.low
        INTERVAL-SEARCH(x.left, i)
    else if x.right != null
        INTERVAL-SEARCH(x.right, i)
```

习题：**14-2**

（Josephus 排列） 定义 Josephus 问题如下：假设 n 个人围成一个圆圈，给定一个正整数 m 且 $m \leqslant n$。从某个指定的人开始，沿环将遇到的每第 m 个人移出队伍。每个人移出之后，继续沿环数剩下来的人。这个过程直到所有的 n 个人都被移出后结束。每个人移出的次序定义了一个来自整数 1，2，…，n 的 (n, m)-Josephus 排列。例如，(7，3)-Josephus 排列为〈3，6，2，7，5，1，4〉。

a. 假设 m 是常数，描述一个 $O(n)$ 时间的算法，使得对于给定的 n，能够输出 (n, m)-Josephus 排列。

b. 假设 m 不是常数，描述一个 $O(n \lg n)$ 时间的算法，使得对于给定的 n，能够输出 (n, m)-Josephus 排列。

第一问：用链表直接模拟即可（或者数组+同余）

第二问："如果你刚刚打印了第k个值，其秩为r，则从树中删除该节点，那么要打印的第(k+ 1)值的秩为 r−1 + m mod (n−k)。由于删除和查找需要O(lg n)，且有n个节点，因此运行时间为O(n lg n)。"

实际上只利用了OS树的 log 性质和有序性……但是解决的问题却扩大了，这就是数据结构的精妙吧！把数组扔到OSTree上！

Ch15

动态规划方法通常用来求解最优化问题(optimization problem)，我们希望寻找具有最优值（最小值或最大值）的解。我们称这样的解为问题的**一个**最优解(an optimal solution)，而不是最优解(the optimal solution)，因为可能有多个解都达到最优值。

用动态规划方法求解的最优化问题应该具备的两个要素：最优子结构和子问题重叠。

最优子结构(optimal substructure)性质：问题的最优解由相关子问题的最优解组合而成，而这些子问题可以独立求解。-> 从而在 dp 过程中，每个（子）问题最多被解决一次，显著节约时间。若满足最优子结构，子问题一定互相无关（互不影响）。

钢管切割问题：

```
CUT-ROD(p, n)
    let v[0..n] be a new array
    v[0] = 0
    for j = 1 to n
        q = -inf
        for i = 1 to j
            q = max(q, p[i] + r[j-i])
        r[j] = q
    return r[n]
```

最长公共子序列问题：

最优子结构：$$c[i,j]=\begin{cases}0 & \text{若 } i=0 \text{ 或 } j=0\\ c[i-1,j-1]+1 & \text{若 } i,j>0 \text{ 且 } x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & \text{若 } i,j>0 \text{ 且 } x_i\neq y_j\end{cases}$$

矩阵链括号划分 和 最优二叉搜索树：

类似Floyd，这样记就行了

习题：

15-12 15-12 签约棒球自由球员

15-11 15-11 库存规划 （较难）

15-10 15-10 投资策略规划

15-9 15-9 字符串拆分（类似Floyd，也可以递归）

15-3 15-3 双调欧几里得旅行商问题

15-4 15-4 整齐打印（较难）

15-5 15-5 编辑距离

15-6 15-6 公司聚会计划（优雅的树上DP）

Ch16

> 贪心算法通常都是这种自顶向下的设计：做出一个选择，然后求解剩下的那个子问题，而不是自底向上地求解出很多子问题，然后再做出选择。

贪心算法原理

两大性质：最优子结构 + 贪心选择性

一般情况下，我们可以按如下步骤设计贪心算法：

- 将最优化问题转换为这样的形式：对其做出一些选择后，只剩下一个子问题需要求解。
- 证明做出贪心选择后原问题总是存在最优解，即贪心选择总是安全的。
- 证明做出贪心选择后剩余的子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解，这样就得到了最优子结构。

贪心选择性

一个全局最优解可以通过局部最优选择（贪心选择）达到， 不考虑子问题的结果；

动态规划每一步都要做出选择，并且选择依赖于子问题的解(先求解出子问题的解，对解进行选择），所以动态规划要自底向上， 从小问题到大问题；
在贪心算法中，我们总是做出当时看来最佳的选择，然后求解剩下的唯一的子问题。贪心算法进行选择时**可能依赖之前做出的选择，但不依赖任何将来的选择或是子问题的解。**因此贪心算法是自顶向下。

活动安排问题：每次选择结束时间最早的

Huffman 编码：O(nlogn)

> 注意，编码树并不是二叉搜索树，因为叶结点并未有序排列，而且内部结点并不包含字符关键字。

会画哈夫曼树就行。

> （16.3-8）假定一个数据文件由 8位字符组成，其中所有256个字符出现的频率大致相同：最高的频率也低于最低频率的2倍。证明：在此情况下，赫夫曼编码并不比8位固定长度编码更高效。

可以边画边证明，即使用Huffman编码，每个字符也需要8位。

习题：

16.1

Ch17

聚合分析：

利用聚合分析，我们证明对所有n, 一个n个操作的序列**最坏情况下（这里必须是一个比较好的上界）**花费的总时间为T(n)。 因此，在最坏情况下，每个操作的平均代价，或摊还代价为 T(n)/n。

注意，此摊还代价是适用于每个操作的，即使序列中有多种类型的操作也是如此。核算法和势能法，对不同类型的操作可能赋予不同的摊还代价。

- 例1：栈操作

> 通过使用聚合分析，我们考虑整个序列的 n 个操作，可以得到更好的上界。实际上，虽然一个单独的 MULTIPOP 操作可能代价很高，但在一个空栈上执行 n 个 PUSH、POP 和 MULTIPOP 的操作序列，代价至多是 $O(n)$。这是为什么呢？当将一个对象压入栈后，我们至多将其弹出一次。因此，对一个非空的栈，可以执行的 POP 操作的次数（包括了 MULTIPOP 中调用 POP 的次数）最多与 PUSH 操作的次数相当，即最多 n 次。因此，对任意的 n 值，任意一个由 n 个 PUSH、POP 和 MULTIPOP 组成的操作序列，最多花费 $O(n)$ 时间。一个操作的平均时间为 $O(n)/n=O(1)$。在聚合分析中，我们将每个操作的摊还代价设定为平均代价。因此，在此例中，所有三种栈操作的摊还代价都是 $O(1)$。
>
> 再次强调，虽然我们已经证明一个栈操作的平均代价，也就是平均运行时间为 $O(1)$，但并未使用概率分析。我们实际上得出的是一个 n 个操作的序列的最坏情况运行时间 $O(n)$，再除以 n 得到了每个操作的平均代价，或者说摊还代价。

- 例2：二进制计数器递增

> 对于 n 个 INCREMENT 操作组成的序列，我们可以得到一个更紧的界——最坏情况下代价为 $O(n)$，因为不可能每次 INCREMENT 操作都翻转所有的二进制位。如图 17-2 所示，每次调用 INCREMENT 时 $A[0]$ 确实都会翻转。而下一位 $A[1]$，则只是每两次调用翻转一次，这样，对一个初值为 0 的计数器执行一个 n 个 INCREMENT 操作的序列，只会使 $A[1]$ 翻转 $\lfloor n/2 \rfloor$ 次。类似地，$A[2]$ 每 4 次调用才翻转一次，即执行一个 n 个 INCREMENT 操作的序列的过程中翻转 $\lfloor n/4 \rfloor$ 次。一般地，对一个初值为 0 的计数器，在执行一个由 n 个 INCREMENT 操作组成的序列的过程中，$A[i]$ 会翻转 $\lfloor n/2^i \rfloor$ 次（$i=0, 1, \cdots, k-1$）。对 $i \geqslant k$，$A[i]$ 不存在，因此也就不会翻转。因此，由公式(A. 6)知，在执行 INCREMENT 序列的过程中进行的翻转操作的总数为
>
> $$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$
>
> 因此，对一个初值为 0 的计数器，执行一个 n 个 INCREMENT 操作的序列的最坏情况时间为 $O(n)$。每个操作的平均代价，即摊还代价为 $O(n)/n=O(1)$。

核算法：

用核算法(accounting method)进行摊还分析时，我们对不同操作赋予不同费用，赋予某些操作的费用可能多于或少于其实际代价。我们将**赋予一个操作的费用称为它的摊还代价**。当一个 操作的摊还代价超出其实际代价时，我们将差额存入数据结构中的特定对象，存入的差额称为**信用**。对于后续操作中摊还代价小于实际代价的情况，信用可以用来支付差额。

例如栈操作中，赋予 PUSH 2 POP 0，而不是 PUSH 1 POP 1. （理解就好……）

势能法：

思想类似于核算法。每个操作的摊还代价=实际代价+此操作引起的势能变化。

势能法工作方式如下。我们将对一个初始数据结构 D_0 执行 n 个操作。对每个 $i=1，2，\cdots，n$，令 c_i 为第 i 个操作的实际代价，令 D_i 为在数据结构 D_{i-1} 上执行第 i 个操作得到的结果数据结构。势函数 Φ 将每个数据结构 D_i 映射到一个实数 $\Phi(D_i)$，此值即为关联到数据结构 D_i 的势。第 i 个操作的摊还代价 $\hat{c}_i$ 用势函数 Φ 定义为：

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{17.2}$$

n 个操作的总摊还代价为：

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

要点是如何定义势函数。

栈操作

为了展示势能法，我们再次回到栈操作 PUSH、POP 和 MULTIPOP 的例子。我们将一个栈的势函数定义为其中的对象数量。对于初始的空栈 D_0，我们有 $\Phi(D_0)=0$。由于栈中对象数目永远不可能为负，因此，第 i 步操作得到的栈 D_i 具有非负的势，即

$$\Phi(D_i) \geqslant 0 = \Phi(D_0)$$

因此，用 Φ 定义的 n 个操作的总摊还代价即为实际代价的一个上界。

下面计算不同栈操作的摊还代价。如果第 i 个操作是 PUSH 操作，此时栈中包含 s 个对象，则势差为

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$

则由公式(17.2)，PUSH 操作的摊还代价为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

假设第 i 个操作是 MULTIPOP(S，k)，将 $k'=\min(k，s)$个对象弹出栈。对象的实际代价为 k'，势差为

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'$$

因此，MULTIPOP 的摊还代价为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$

类似地，普通 POP 操作的摊还代价也为 0。

每个操作的摊还代价都是 $O(1)$，因此，n 个操作的总摊还代价为 $O(n)$。由于我们已经论证了 $\Phi(D_i)\geqslant\Phi(D_0)$，因此，$n$ 个操作的总摊还代价为总实际代价的上界。所以 n 个操作的最坏情况时间为 $O(n)$。

Ch19（本章没留作业）

一个斐波那契堆是一系列具有**最小堆序**(min-heap ordered)的有根树的集合。也就是说，每棵树均遵循**最小堆性质**(min-heap property)：每个结点的关键字大于或等于它的父结点的关键字。图 19-2(a)是一个斐波那契堆的例子。

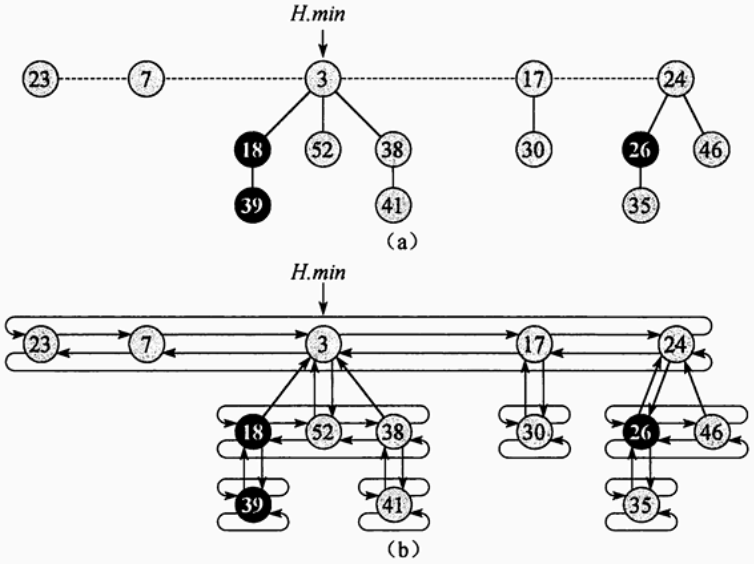


图 19-2 (a)一个包含 5 棵最小堆序树和 14 个结点的斐波那契堆。虚线标出了根链表。堆中最小的结点是包含关键字 3 的结点。黑色的结点是被标记的。这个斐波那契堆的势是5+2×3=11。(b)一个更加完整的表示，显示出了指针 p(向上箭头)、$child$(向下箭头)、$left$ 和 $right$(横向箭头)。本章剩下的图省略了这些细节，因为该图中显示的所有信息均可从(a)图中推断出来

一些定义：

孩子链表：x 的孩子链表是 x 的所有孩子被链接形成的双向链表

x.child：仅指向 x 的某一个孩子

x.degree：x 的孩子数目

x.mark：“指示结点x 自从上一次成为另一个结点的孩子后，是否失去过孩子”

H.min ： 具有最小关键字的树的根节点，也就是斐波那契堆的“最小结点”

H.n ： H 中当前的结点数目

根链表：所有树的根形成的双向链表

势函数：$\Phi H = tree(H) + marked(H)$ = 树的数目+已标记的结点数

Ch21

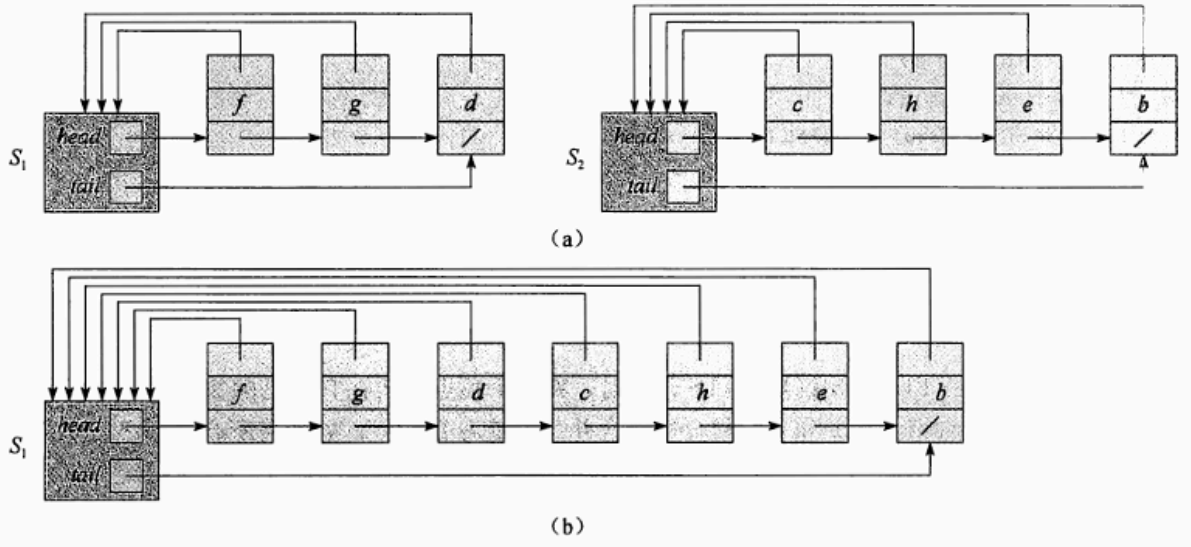
确定连通分量（朴素）：

```
CONNECTED-COMPONENTS(G)
    for each v in G.V
        MAKE-SET(v)
    for each edge(u, v) in G.E
        if FIND-SET(u) != FIND-SET(v)
            UNION(u, v)

SAME-COMPONENT(u, v)
    if FIND-SET(u) == FIND-SET(v)
        return true
    return false
```

用链表表示：

用这种链表表示，MAKE-SET 操作和 FIND-SET 操作是非常方便的，只需 $O(1)$ 的时间。要执行 MAKE-SET(x)操作，我们需要创建一个只有 x 对象的新的链表。对于 FIND-SET(x)，仅沿着 x 对象的返回指针返回到集合对象，然后返回 *head* 指向对象的成员。例如，在图 21-2(a)中，FIND-SET(g)的调用将返回 f。



定理 21.1 使用不相交集合的链表表示和加权合并启发式策略，一个具有 m 个 MAKE-SET、UNION 和 FIND-SET 操作的序列(其中有 n 个是 MAKE-SET 操作)需要的时间为 $O(m+n\lg n)$。

用森林表示：

两种启发式合并策略：**按秩合并，路径压缩**

```
MAKE-SET(x)
    x.p = x
    x.rank = 0

UNION(x, y)  // 每次 UNION 至多减少一个连通分量
    LINK(FIND-SET(x), FIND-SET(y))

FIND-SET(x) // 实际就是冰茶几
    if x != x.p
        x.p = FIND-set(x, p)
    return x.p

LINK(x, y)
```

```
    if x.rank > y.rank
        y.p = x
    else
        x.p = y
        if x.rank == y.rank
            y.rank = y.rank + 1
```

> FIND-SET 过程是一种 **两趟方法**

每个 MAKE-SET 操作的摊还代价为 $O(1)$

每个 LINK 操作的摊还代价为 $O(\alpha(n))$

每个 FIND-SET 操作的摊还代价为 $O(\alpha(n))$

$$\alpha(n)=\begin{cases}0 & \text{对 } 0\leqslant n\leqslant 2\\ 1 & \text{对 } n=3\\ 2 & \text{对 } 4\leqslant n\leqslant 7\\ 3 & \text{对 } 8\leqslant n\leqslant 2\,047\\ 4 & \text{对 } 2\,048\leqslant n\leqslant A_4(1)\end{cases}$$

Ch22 23

带时间戳的DFS：

```
DFS(G)
 1  for each vertex u ∈ G.V
 2      u.color = WHITE
 3      u.π = NIL
 4  time = 0
 5  for each vertex u ∈ G.V
 6      if u.color == WHITE
 7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

定理 22.7（括号化定理） 在对有向或无向图 $G=(V,E)$ 进行的任意深度优先搜索中，对于任意两个结点 u 和 v 来说，下面三种情况只有一种成立：

- 区间 $[u.d, u.f]$ 和区间 $[v.d, v.f]$ 完全分离，在深度优先森林中，结点 u 不是结点 v 的后代，结点 v 也不是结点 u 的后代。
- 区间 $[u.d, u.f]$ 完全包含在区间 $[v.d, v.f]$ 内，在深度优先树中，结点 u 是结点 v 的后代。
- 区间 $[v.d, v.f]$ 完全包含在区间 $[u.d, u.f]$ 内，在深度优先树中，结点 v 是结点 u 的后代。

（要么完全包含，要么相离）

拓扑排序：

对于一个有向无环图 G= (V,E) 来说，其拓扑排序是 G 中所有结点的一种**线性次序**，该次序满足如下条件：如果图 G 包含边 (u, v), 则结点 u 在拓扑排序中处于结点 v 的前面（如果图 G 包含环路，则不可能排出一个线性次序）。

课本给出的拓扑排序方法是先DFS，再将各点按照 u.f (完成时间) 的逆序排列。

最小生成树：

在本章中，我们将详细讨论解决最小生成树问题的两种算法：Kruskal 算法和 Prim 算法。如果使用普通的二叉堆，那么可以很容易地将这两个算法的时间复杂度限制在 $O(E \lg V)$ 的数量级内。但如果使用斐波那契堆，Prim 算法的运行时间将改善为 $O(E+V \lg V)$。此运行时间在 $|V|$ 远远小于 $|E|$ 的情况下较二叉堆有很大改进。

Ch25（实际上没讲）

Johnson 多源最短路：Bellman-Ford + Reweighting + Dijkstra

Johnson算法具体步骤（翻译自wikipedia）：

『
1.初始化，把一个node q添加到图G中，使node q 到图G每一个点的权值为0。
2.使用Bellman-Ford算法，从源点为q，寻找每一个点 v从q到v的最短路径h(v)，如果存在负环的话，算法终止。
3.使用第2步骤中Bellman-Ford计算的最短路径值对原来的图进行reweight操作（重赋值）：边<u,v>的权值w(u,v)，修改成w(u,v)+h(u)-h(v)。
4.最后，移去q，针对新图（重赋值之后的图）使用Dijkstra算法计算从每一个点s到其余另外点的最短距离。
』[10]

会画图就行

Ch31

gcd：

欧几里得算法：

```
EUCLID(a, b)
    if b == 0
        return a
    else return EUCLID(b, a mod b)
```

扩展欧几里得算法：

```
EXTENDED-EUCLID(a, b)
    if b == 0
        return (a, 1, 0)
    else
        (d1, x1, y1) = EXTEND-EUCLID(b, a mod b)
        (d, x, y) = (d1, y1, x1 - (a/b)y1)
        return (d, x, y)
```

欧几里得算法和扩展欧几里得算法中，递归调用次数约为 $O(logb)$

模线性方程：

解存在的条件：

推论 31.21 当且仅当 $d|b$ 时，方程 $ax \equiv b(\mathrm{mod}\, n)$ 对于未知量 x 有解，这里 $d = \gcd(a, n)$。

若解存在，则在模意义下有 d 个不同的解。

用扩展欧几里得算法求解：

```
MODULAR-LINEAR-EQUATION-SOLVER
    (d, x, y) = EXTENDED-EUCLID(a, n)
    if !(d|b) then return -1
    x0 = (x * b / d) mod n
    for i = 0 to n - 1
        print (x0 + i*n/d) mod n
```

中国剩余定理解同余方程组：

同余方程组的形式应为 $x \equiv a_i \ (mod\ n_i)$ 且 n_i 与 n_j 互质

- 计算 M、M_i
- 计算 M_i 在模 n_i 意义下的乘法逆元 M_i^{-1}
- 计算 $c_i = M_i \times (M_i^{-1}\ mod\ n_i)$
- 答案 $a = \Sigma\ c_i \times a_i \ (mod\ M)$

简单素数测试算法：

试着用每个整数 2，3，…，$\lfloor\sqrt{n}\rfloor$ 分别去除 n。

最坏运行时间是 $\Theta(\sqrt{n}) = \Theta(2^{\beta/2})$，其中 β 是 n 二进制表示后的位数。

伪素数测试算法：

如果 n 是一个合数，且 $a^{n-1} \equiv 1 \ (mod \ n)$，则称 n 是一个基为 a 的伪素数。

下面的算法采用基为2的伪素数测试，如果上述等式不成立，n 必然是合数；如果上述等式成立，n 要么是素数，要么是基为 2 的伪素数。

```
PSEUDO-PRIME(n)
    if POW(2, n-1) mod n != 1
        return COMPOSITE
    else return PRIME
```

错判合数为素数：Carmichael 数（561、1105、1729.....）

Miller-Rabin 算法：

改进伪素数测试算法，使得 Carmichael 数 不会被错判；时间复杂度为 $O(TlogN)$ ，T 是测试基的轮数。

原理：浅谈Miller-Rabin素数检测算法 - Seaway-Fu - 博客园

> 介绍下一个理论依据：**二次探测定理。**
>
> 二次探测定理的内容是：
>
> 如果一个数 p 是质数，对于一个 $x \in (0,p)$且$x \in \mathbb{Z}$，方程 $x^2 \equiv 1(mod \ p)$的解有且只有两个：$x=1$ 或 $x=p-1$。
>
> 那么，在快速幂累乘的基础上，反复判断现在的 P 是否符合二次探测定理，就大大增加了其正确性。
>
> 具体的实现方式是：
>
> 对于伪素数测试之中的指数 $n-1$，我们可以将其分解成 $m \times 2^k$ 的形式，其中 m 为奇数。那么根据快速幂的原理，我们就会依次对以下数列进行二次探测定理的检测：
>
> $$m, 2m, 4m \cdots m \times 2^{k-1}, m \times 2^k$$
>
> 如果这些都合法，最后用费马小定理判断一下是否合法即可。

两个改进：

Miller-Rabin 素数测试方法对简单测试过程 PSFUDOPRIME 做了两点改进，克服了其中存在的问题：

- 它试验了多个随机选取的基值 n，而非仅仅一个基值。
- 当计算每个模取幂的值时，在最后一组平方里，寻找一个以 n 为模的 1 的非平凡平方根。如果发现一个，终止执行并输出结果 COMPOSITE。31.6 节的推论 31.35 证明了用这种方法检测合数的正确性。

```
bool Miller_check(int a,int n)
{
    int ret=1;
    int b=n-1;
    while(b)
    {
        if(b&1)
```

```
            ret=(ret*a)%n;
        int x=a;//采用临时变量保存改变前的a
        a=(a*a)%n;
        if(a==1 && x!=1 && x!=n-1)//当a为1的时候，方程成立，开始判断解。
            return 0;//不是素数
        b>>=1;
    }
    return (ret==1)?1:0;
}
bool Miller_Rabin(int n,int t)
{
    if(n==2)
        return 1;
    if(n<2 || !(n&1))
        return 0;
    while(t--)
    {
        srand(time(NULL));
        int a=rand();//a要随机数
        if(!Miller_check(a,n))
            return 0;
    }
    return 1;
}
```

其他：

RSA 加密系统的安全性主要来源于对大整数进行因式分解的困难性。

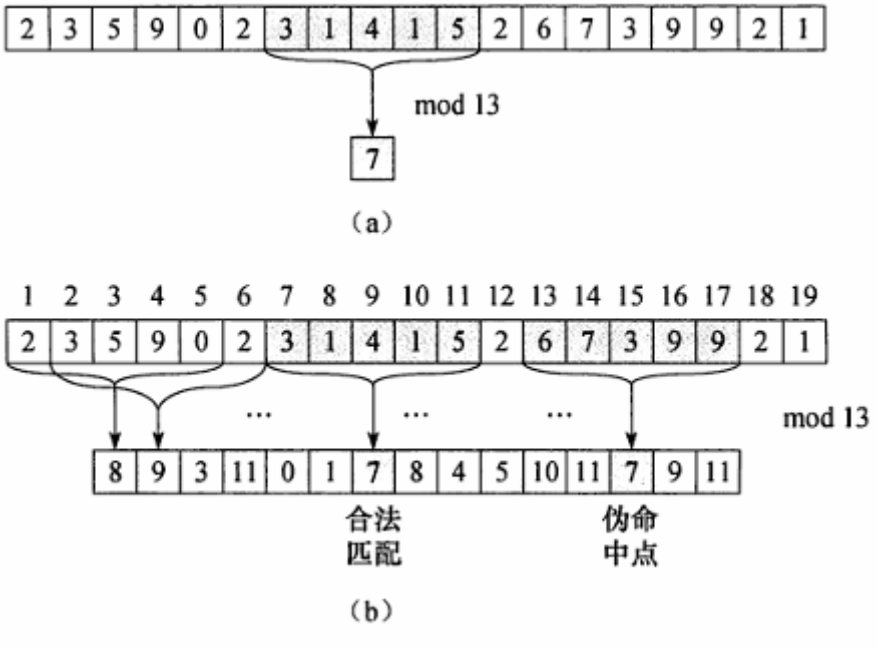
Ch32

算　法	预处理时间	匹配时间
朴素算法	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
有限自动机算法	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

图 32-2　本章的字符串匹配算法及其预处理时间和匹配时间

Rabin-Karp 串匹配算法：

类似于哈希



模数较大时期望运行时间为 $O(n)+O(m(v+n/q))\sim O(n)+O(m)\sim O(n)$

有限自动机串匹配算法：

会画图、理解字符表 Σ 的含义就行

KMP：

模式串匹配自身得到前后缀数组，模式串再与文本串匹配

```
KMP(T, P)
    n = T.length
    m = P.length
    prefix = COMPUTE_PREFIX(P)
    q = 0
    for i = 1 to n
        while q>0 and P[q+1]!=T[i]
            q = prefix[q]
        if P[q+1]==T[i]
            q = q + 1
        if q == m
            return true
        q = prefix[q]
```

```
COMPUTE_PREFIX(P)
    m = p.length
    let prefix[1..m] be a new array
    prefix[1] = 0
    k = 0
    for q = 2 to m
        while k>0 and P[k+1]!=P[q]
            k = prefix[k]
        if P[k+1]==P[q]
            k = k + 1
        prefix[q] = k
    return prefix
```