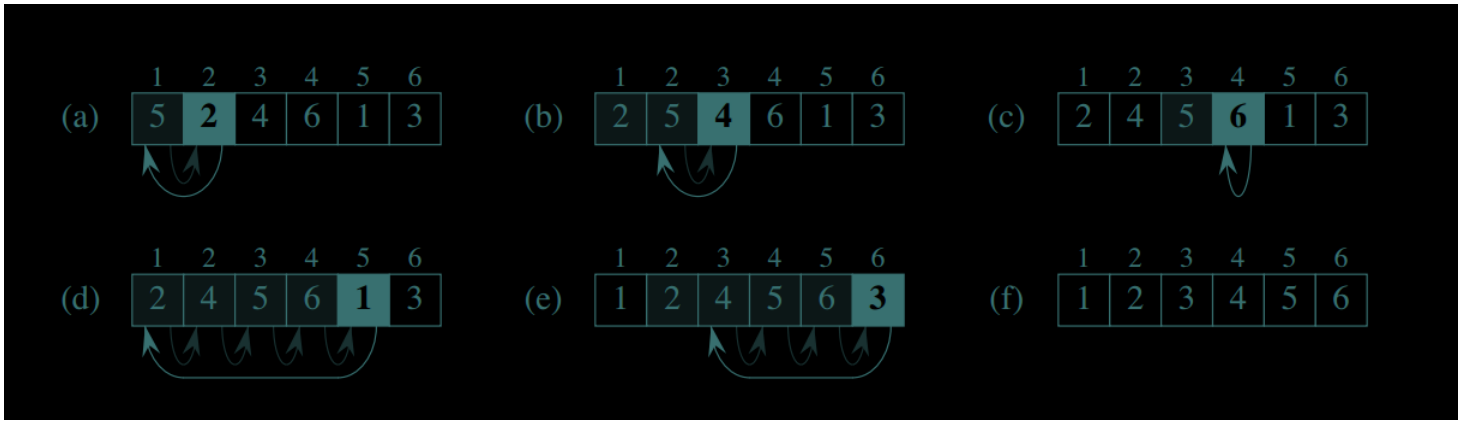
② Getting Started

生词	释义
outperform	胜过
sentinel	哨兵
auxiliary	辅助的
paradigm	范例，典型
intuitively	直观地
recursive	回归的，递归的
concise	简明的
tedious	乏味的
inferoir	差的，下等的
viable	可行的，能生育的
attribute	属性
substitute	取代
pseudocode	伪代码
modularity	模块化
concisely	简明地
invariant	不变的，不变量
trivially	琐碎地，很一般地，不重要地

- 本章首先介绍插入排序算法，然后对算法进行分析，主要是运行时间。然后介绍分治方法，使用这一思想得到归并算法，最后分析归并排序的运行时间。

2.1 插入排序

- 插入算法是一种对小数量元素很有效率的排序算法。举个例子来说，插入排序很像是我们打扑克时排列手牌的方法，假设一摞牌面朝下放在桌子上，你一张一张地拾起来然后放入手中排好序的牌组中，你每拿起一张，就从左到右比较手中的每一张牌，然后放在合适的位置，知道所有的牌都进入手中。



插入排序代码：

```
void insertion_sort(struct array *array)
{
    int key, i;
    for(int j = 1; j < array->used; j++)
    {
        key = array->arr[j];
        for(i = j - 1; i >= 0 && array->arr[i] > key; i--)
        {
            array->arr[i+1] = array->arr[i];
        }
        array->arr[i+1] = key;
    }
}
```

- 这段代码中有一个重要的概念：循环不变量。在这段代码中体现为 array->arr[0] 到 array->arr[j-1]。也就是在每次循环中，这段数列中的元素是不变的，仅仅是顺序发生了变化。
- 循环不变量可以用来检测代码的正确性，引入类似与数学归纳法的概念，通过三个步骤：
  1. 初始：在循环开始时循环不变量符合算法要求；
  2. 保持：如果在某次循环之前循环不变量满足要求，则循环之后仍然满足要求；
  3. 终止：在终止时，检验循环是否覆盖到了所有该做的事情，换句话说就是循环不变量是否在终止时增长为整个数组。

  在 Insertion_sort 代码中，显然就可以用这三个步骤很简单地证明代码的正确性。

下面复习一点编程的知识

- 在for循环一定要记住，循环计数量最后的值一定是第一个超出判定界限的那个值，这个值在循环结束后会保留到后面的代码中。。

- 从现在开始要把表示对象或数组的量看作 指针 了，几个例子，比如把两个对象相等：x = y，那么这两个指针就会指向同一个对象，那么改变了 y 的属性必然会使 x 的属性也改变。
- 过程的参数传递 是个重要的问题，一定要注意，如果传递给“被调用过程”的参数是一个值，那么“调用过程”则看不到“被调用过程”对其进行的赋值；如果传递的是一个对象，则指向对象的指针被复制，而对象的属性不被复制，也就是说对这个参数本身进行赋值，则“调用过程”看不到，而对这个参数的属性进行复制，“调用过程”看得到。
- return 将立刻将控制返回到“调用过程”的调用点。
- bool 表示式都是“短路的”。也就是说，一旦已经计算的值已经可以确定 bool 表示式的值的时候，就不继续往下计算了。
- 调用过程来处理被调用过程的error。

练习

2. 升降序的函数都写了，参见代码。
3. search函数在代码中已经写了，重点是学会用循环不变性来检测代码的正确性，对于循环不变量 A[0] to A[i]：
    1. 初始时 i = 0，循环不变量中只有一个元素，在循环的第一步进行比对，所以初始元素已经排查；
    2. 在循环过程中，如果循环之前循环不变量已经经过排查，那么循环之后则完成对“新”元素的排查，所以也满足要求。
    3. 在终止时，i = array->used，所以覆盖到了数列所有的元素，结果正确。
4. 在 array.c 中已经实现了二进制数的加法。

2.2 分析算法

- 分析一个算法就是预测这个算法运行所需要的资源。其中最为关心的就是时间这个资源。
- 在本书中，用于分析算法的基本模型名叫 RAM（随机访问器），是一个单处理器系统。在这个模型中指令一条一条运行，并没有并行。
- 我们的 RAM，包含所有的基本功能如下
    1. 算术指令：加减乘除、取余、向上下取整；
    2. 数据移动：加载、储存、拷贝；
    3. 控制：条件&无条件转移、子程序调用、返回。

  所有上述指令都耗费固定的时间。另外在这个模型中并没有考虑内存层级架构（除了为数不多的几个例子外），是因为那样做太过于复杂，这样的模型已经能很好的进行算法分析了。

- 在分析之前，必须明确定义“输入规模”和“运行时间”两个概念，对于不同的问题有不同的定义，所以需要针对不同问题单独指出。运行时间一般用执行的基本操作数或者步数来度量。

INSERTION-SORT(A)	代价	次数
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

我们假定第 i 步指令运行所需的时间为常量 C_i，那么总的运行时间可以使用上式求出，注意到式中有 T_j，它代表这个语句在 j 次循环中运行的次数，这个值有可能会因输入的不同而不同，也就是输入的情况。在最好的情况下，结果为

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

可以表示为线性 an + b 的形式。在最差的情况下，结果为

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

可以表示为二次函数 an^2 + bn + c 的形式。

- 在本书的余下部分，我们主要求 最差情况下的运行时间。理由是：
  1. 最长的运行时间标定了一个上界，这是很重要的。
  2. 对某些算法，最坏情况很容易出现，比如说检索一个数据库中没有的信息。
  3. “平均运行时间”很有可能与最差情况在一个增长量级上。
- 某些时候，我们会用 随机化算法，来对算法的运行时间进行一个概率分析。这将在第5章及后续章节进行讲述。

- 从上述对运行时间的分析可看出这是个十分繁杂的过程，实际的分析过程必须进行简化，引入 增长量级 的概念，我们忽略低次项只关注 最高次项，并把常数因子忽略。比如插入排序算法的增长量级就是 $\Theta(n^2)$。
- 但是低的增长量级的算法在相对小的输入规模时未必就一定比更高增长量级的算法用时少！因为这是忽略了常数因子和低次项的简化结果，只有在输入规模 n 足够大时才成立。

练习

- 2.2.1
  $\Theta(n^3)$
- 2.2.2
  selection_sort 已经在代码中实现，下面进行增长量级分析：

```
void selection_sort(p1* array)
{
    int key, i, tem;
    for(int j = 0; j < array->used - 1; j++)
    {
        key = j;
        for(i = j + 1; i < array->used; i++)
        {
            if(array->arr[i] < array->arr[key])
                key = i;
        }
        tem = array->arr[key];
        array->arr[key] = array->arr[j];
        array->arr[j] = tem;
    }
}
```

代价 次数
C1 1
C2 n
C3 n-1
C4 n(n+1)/2
C5 n(n-1)/2
C6 (n^2-1)/4, 奇
C7 n-1
C8 n-1
C9 n-1
(n^2)/4, 偶

其中 C6 项的次数比较有趣，我只是猜测最差情况为倒序输入，但实际情况是否正确我没有证明。这次分析使用了最复杂的方式，即逐语句分析代价和运行次数，实际上观察代码，运行次数最多的语句一定是嵌套的 for 语句，所以增长量级就是 $\Theta(n^2)$，前提条件符合所有语句都耗费有限的时间。

- 2.2.3
  通过概率的分析，对于搜索元素等可能存在数列中任何位置的条件下，规模为 n 的输入平均搜索次数为 $\frac{n+1}{2}$，最坏情况下的搜索次数为 n，所以这个算法的增长量级为 $\Theta(n)$。
  -2.2.4
  那就是根据最好的输入情况去“定制”代码咯，最简单的方法就是（答案上提供的），检测输入是否

为最佳输入，如果是的话就输入一个预先计算好的答案，但是这个检测过程的运行也需要考虑在内，不能得不偿失。

2.3 设计算法

上面所述的插入排序以及选择排序，都是基于一种 增量 的方法。本节我们介绍一种全新的思路，叫做 分治。简而言之就是“分而治之”，大问题化成小问题，逐个解决小问题，大问题也就解决了。我们将在第四章详细讲述分治法，使用分支法设计的排序算法将会比插入排序有更短的最长运行时间。

2.3.1 分治法

- 分治的思想是：把问题分成多个与原问题相似但规模较小的子问题，一直向下分直到子问题的规模足够小，然后使用递归的方法 逐级 解决这些子问题，将结果结合起来形成最终答案。
- 合并排序法是使用分治法的范例：
  - 分：把一个 n 元素的数列分为两个 n/2 规模的子数列；
  - 治：递归地，对每个子数列使用合并排序进行排序；
  - 并：将两个排好序的子数列合并。

  关键的操作是 合并 步，考虑一个称为 $MERGE(A,p,q,r)$ 的过程，A 是一个数列，p，q，r 都是数列中的下标，满足 $p<q<r$，并且 [p, q] 和 [q + 1, r] 的元素都是已经排序的，这个过程复杂嘛？其实这是一个很简单的 $\Theta(n)$，$n=r-p+1$ 的过程，具体做法是这样的，就是想象有两个牌堆，每一张都是面朝上的，然后每次都从堆顶选择小的那张放到一个新的堆，重复这样的过程，直到一个牌堆空了，然后把另一个剩下的牌全部放到新牌堆，就这样，新牌堆就是排好序了。伪代码如下：

```
MERGE(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

- 现在归并算法还没有完成，上述过程只能作为归并算法中的一个子程序，下面将归并算法完成：

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

作为其中重要的一步，如何将一个数列一分为二很重要，这里 $q = \lfloor (p + r)/2 \rfloor$ 需要解释一下，这样设置 q 的值能产生规模分别为 $\lceil \frac{n}{2} \rceil$ 和 $\lfloor \frac{n}{2} \rfloor$ 的两个子数列，最简单的证明方法是对 p 和 r 为奇偶的四种情况进行分类讨论。（我真的算了一下...）

- C 语言代码的归并排序已经完成，可在代码页查看。

2.3.2 对分治法的分析

- 当一个算法包含对其自身的递归调用时，往往用 递归方程式 来描述其运行时间。a 表示原问题被拆分成了 a 个子问题，n / b 表示子问题的规模（a 未必等于 b，只有均分的时候才相等），其中 D(n) 是将原问题拆分所用的时间，C(n) 是将子问题的结果合并的时间。c 表示一个有限的规模，输入在这个规模之下则将运行时间看作是一个常量。

$$T(n) = \begin{cases} \Theta(1) & \text{if n} \leq \text{c}, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

- 我们将在第四章讲述如何解出递归方程式。

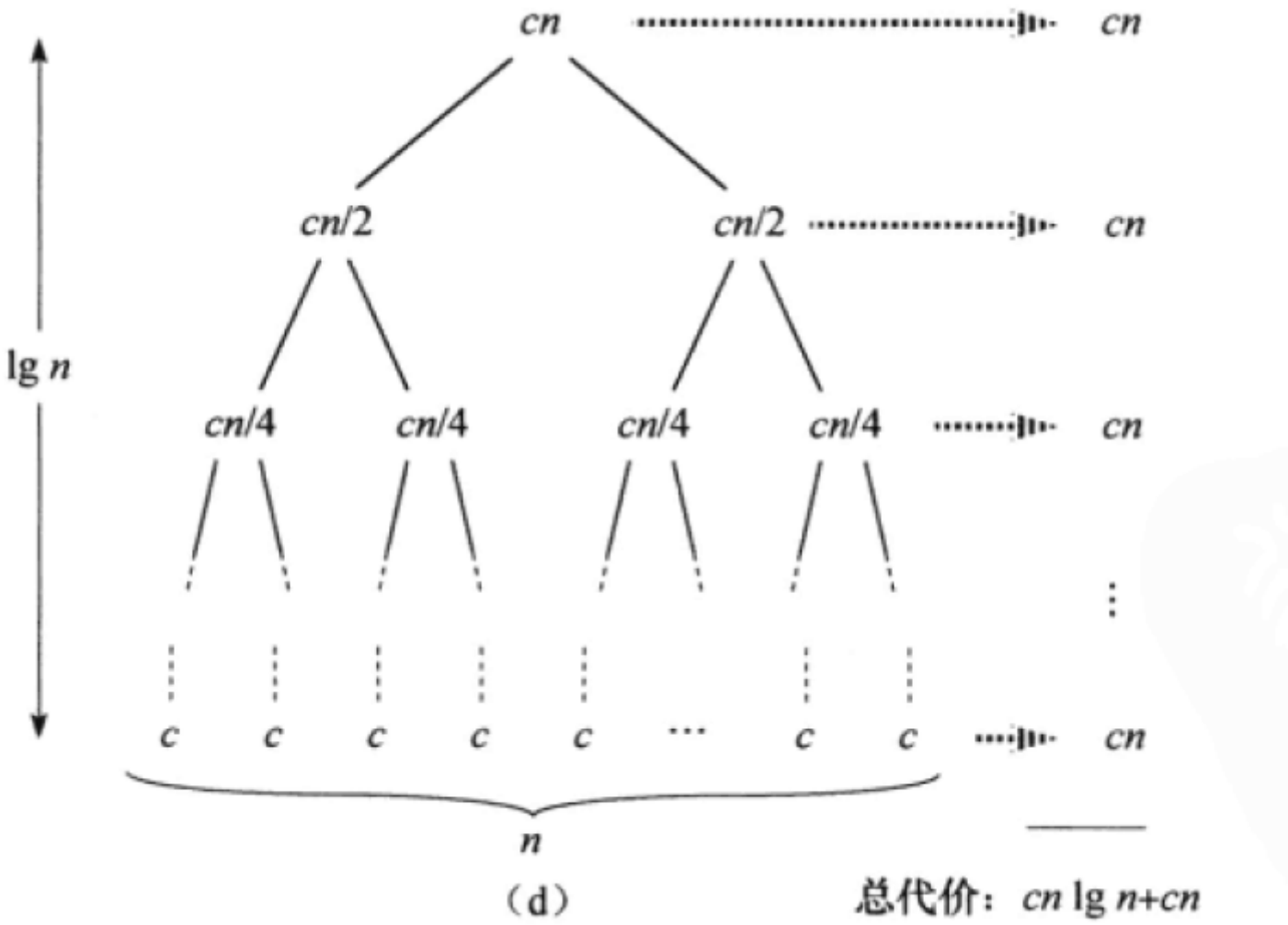
对归并排序的分析

- 我们注意到，假如原问题的规模恰好是 2 的幂的话，那么划分恰可以达到均分，但是并非所有的问题都是 2 的幂，不过，这样假设不影响计算的结果。
- 先假设原问题的输入规模是 2 的幂，根据这样的分析，上式变为：

$$T(n) = \begin{cases} c & \text{n} = 1, \\ 2T(n/2) + cn & \text{n} > 1. \end{cases}$$

相同的常量一般不可能刚好既代表求解规模为 1 的问题的时间又代表在分解步骤与合并步骤处理每个数组元素的时间。通过假设 c 为这两个时间的较大者并认为我们的递归式将给出运行时间的一个上界，或者通过假设 c 为这两个时间的较小者并认为我们的递归式将给出运行时间的一个下界，我们可以回避这个问题。两个界的阶都是 $n \lg n$，合在一起将给出运行时间为 $\Theta(n \lg n)$。

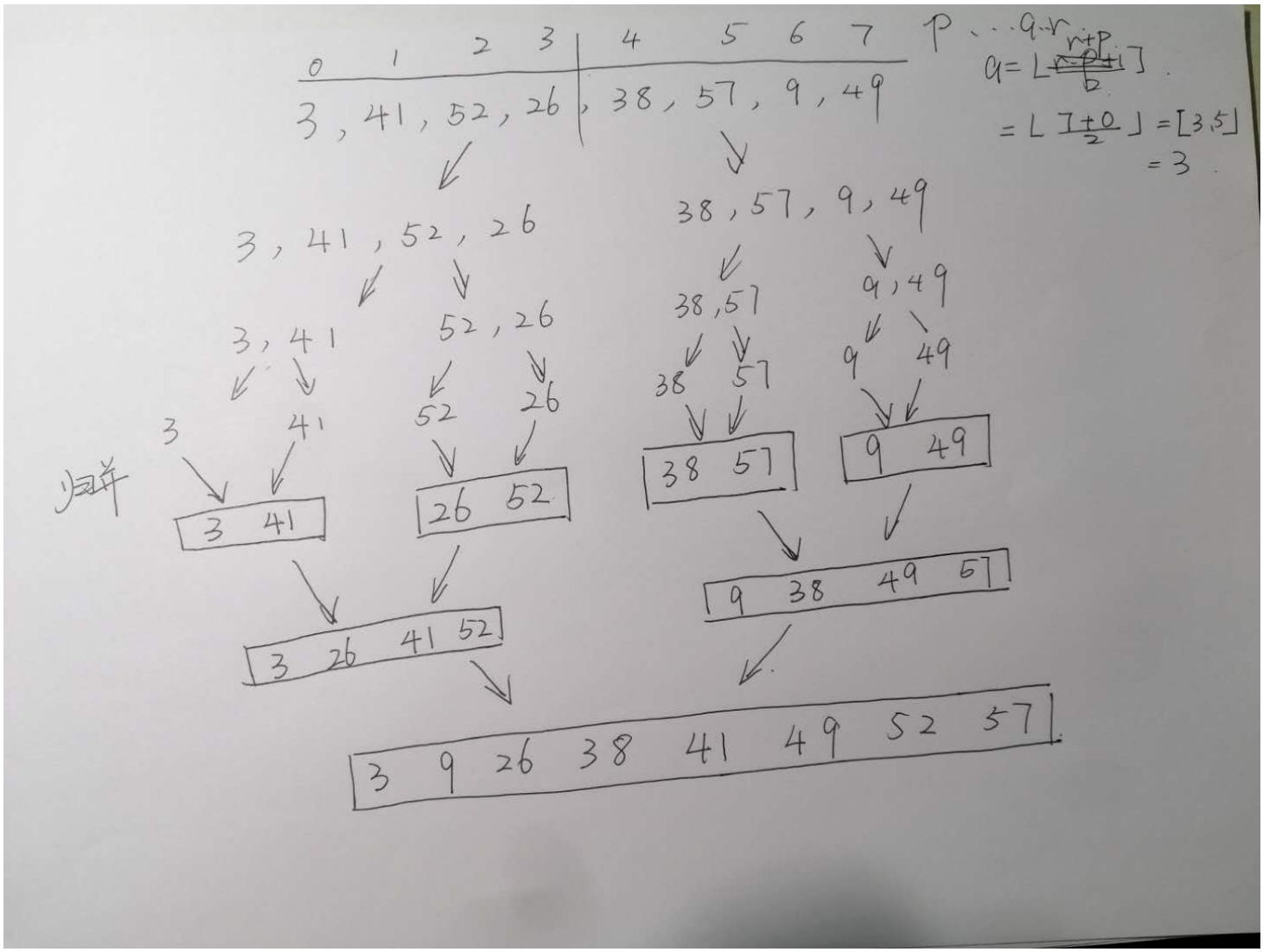
- 被上式这样表达之后，我们就可以进行如下图的划分，称为递归树，n 是叶数：



- 本节这种方法是比较巧妙地，我们将在第四章学习“主定理”，该定理也可直接推得结果。

练习

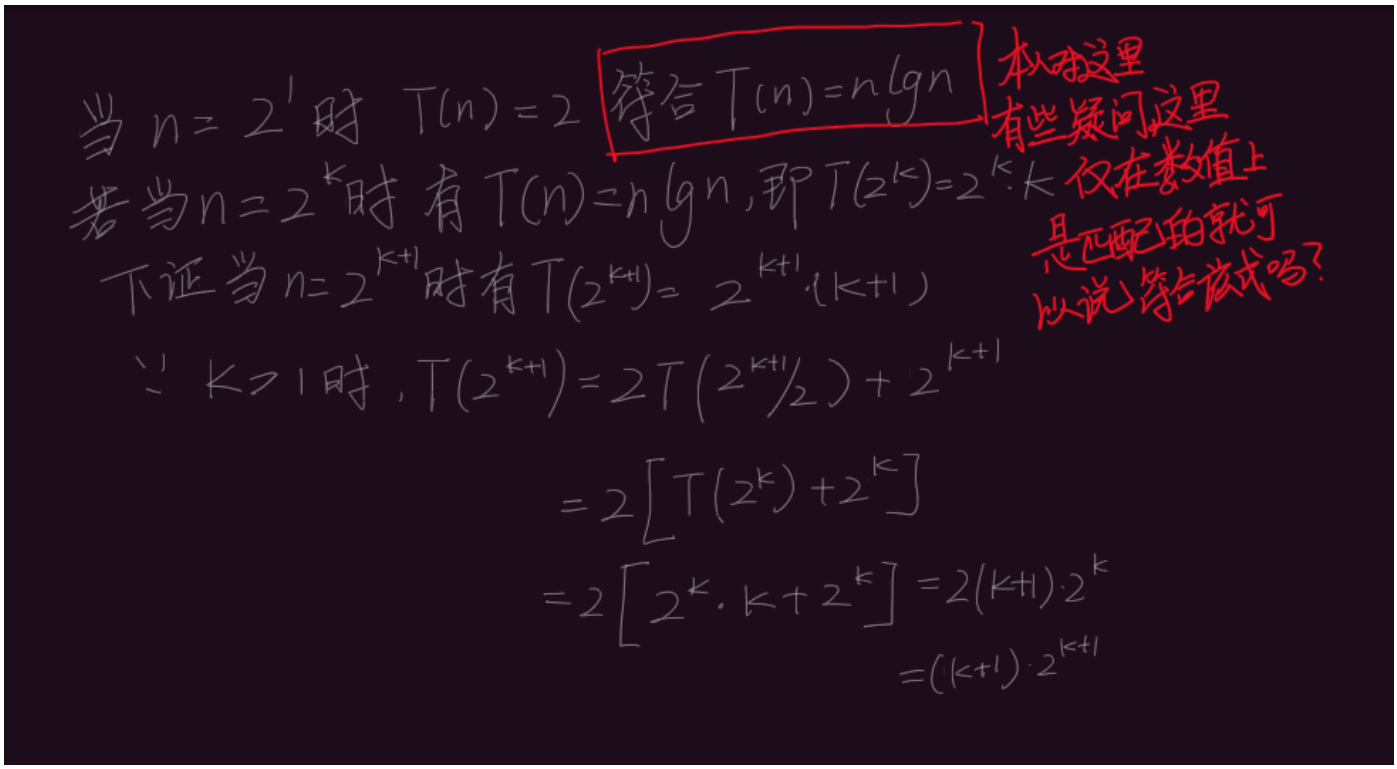
- 2.3.1



- 2.3.2
  如果不用哨兵的话，那就得在两个牌堆都没有抽完之前每次检测任何一个牌堆是否抽完。C 代码如下：

```
void merge(p1* array, int p, int q, int r)
{
    int n1 = q - p + 1;
    int n2 = r - q;
    int* l = (int *)malloc((n1) * sizeof(int));
    int* e = (int *)malloc((n2) * sizeof(int));
    int i, j, k;

    for(i = 0; i < n1; i++)
    {
            l[i] = array->arr[p + i];
    }

    for(j = 1; j < n2 + 1; j++)
    {
            e[j - 1] = array->arr[q + j];
    }

    i = 0;
    j = 0;
    for(k = p; k < r + 1; k++)
    {
            if(l[i] <= e[j])
            {
                    array->arr[k] = l[i];
                    i++;
            }
            else
            {
                    array->arr[k] = e[j];
                    j++;
            }

        if(i == n1)
        {
            for(k++; k < r + 1; k++, j++)
            {
                array->arr[k] = e[j];
            }
            break;
        }
        else if(j == n2)
        {
            for(k++; k < r + 1; k++, i++)
            {
                array->arr[k] = l[i];
            }
            break;
        }
    }
}
```

- 2.3.3



- 2.3.4

```
void recursive_i_sort(p1* array, int t)
{
        if(t == 1)
        {
                return NULL;
        }
        recursive_i_sort(array, t - 1);
        int key, i;
        key = array->arr[t - 1];
        for(i = t - 2; i >= 0 && array->arr[i] > key; i--)
        {
                array->arr[i + 1] = array->arr[i];
        }
        array->arr[i + 1] = key;
        return NULL;
}
```

这个最坏时间的递归式怎么写，老实说我真的不会，答案上是这么写的：

$$T(n) = \begin{cases} \Theta(1) & \text{n} = 1, \\ T(n-1) + \Theta(n) & \text{n} > 1. \end{cases}$$

我也不会计算这个递归式的级数，书上说第四章会讲到。

- 2.3.6

在写本道题之前我就在考虑即便是使用二分法减少了搜索插入位置的时间，那插入的过程仍然是一步一步移动的，这样能使最坏运行时间改进到 nlgn 的级别嘛？

```
void binary_i_sort(struct array *array)
{
        int key, i, low, high, z;
        for(int j = 1; j < array->used; j++)
        {
                key = array->arr[j];
                for(low = 0, high = j - 1; low < high;)
                {
                        int mid;
                        mid = (low + high) / 2;
                        if(key == array->arr[mid])
                        {
                                break;
                        }
                        else if(key < array->arr[mid])
                        {
                                high = mid - 1;
                        }
                        else
                        {
                                low = mid + 1;
                        }
                }
                if(key <= array->arr[low])
                {
                        i = low;
                }
                else
                {
                        i = low + 1;
                }
                for(z = j - 1; z >= i; z--)
                {
                        array->arr[z + 1] = array->arr[z];
                }
                array->arr[i] = key;
        }
}
```

答案上对最坏时间的分析是这样的：在最坏情况下，二分查找的时间复杂度是 nlgn，但插入时数组移动的时间复杂度仍然是 n^2，故总体运行时间不能改善为 $\Theta(nlgn)$。但若排序中使用链表的数据结构，则可以实现。

- 2.3.7

这道题目感觉还是非常有趣的，基本思路如下：先用上一题的二分插入排序（数据结构是链表的），然后针对每一个元素，使用二分查找方法搜索其互补元素。但是我觉得这样不够细节，很多情况下会进行大量不必要的搜索，所以针对这道题，再加上一个范围的限定就更快了。