算法设计与分析作业答案中文版

作者：小道士

版本：2021 年秋季学期

细节是魔鬼，每一道题都值得继续深究。

1 第一次作业

1.1 数组中的第 K 个最大元素

1.1.1 解答：

1. 代码

Algorithm 1 快速查找算法

QUICK-SELECT(nums,left,right,index)

```
q=RANDOM-PARTITION(nums,left,right)
if q==index then
    return nums[q]
else if q>index then
    return QUICK-SELECT(nums,left,q-1,q-index)
end if
return QUICK-SELECT(nums,q+1,right,index)
```

其中 RANDOM-PARTITION(nums,left,right) 定义如下:

Algorithm 2 随机划分算法

RANDOM-PARTITION(nums,left,right)

```
i=RANDOM(left,right)//获取 [left,right] 之间的随机数
exchange nums[right] with nums[i]
x=nums[right]
j=left-1
for m=left to right-1 do
    if nums[m]<=x then
        j=j+1
        exchange nums[j] with nums[m]
    end if
end for
exchange nums[j+1] with nums[right]
return j+1
```

用 $C++$ 代码实现如下:

```
//特别注意加引用！！！
int random_partition(vector<int> &nums, int left ,
int right) {
int i = (rand() % (right - left + 1)) + left;
swap(nums[right], nums[i]);
int x = nums[right];
i = left - 1;
for (int m = left; m < right; m++)
```

```
{
    if (nums[m] <= x)
    {
        i++;
        swap(nums[i], nums[m]);
    }
}
swap(nums[i + 1], nums[right]);
return i + 1;
}
int quick_select(vector<int>& nums, int left,
int right, int index) {
int q = random_partition(nums, left, right);
if (q == index)
    return nums[q];
else if (q > index)
    return quick_select(nums, left, q - 1, index);
return quick_select(nums, q + 1, right, index);
}
```

2. 归约图

3. 正确性

上述的过程中，我们利用快速排序的划分函数对数组进行划分，并根据划分后返回的下标 i 来进行区间缩小的依据。$A[i]$ 左边元素有 $left\_element = i - L + 1$ 个，且左边的元素均比 $A[i]$ 小，而右边的元素比 $A[i]$ 大。

$left\_element == k$：显然 $A[i]$ 就是要找的元素。

$left\_element < k$：左边的元素不够 K 个，说明第 K 个元素在右边。变为查找 $[i+1, R]$ 中的第 $k - left\_element$ 大元素。

$left\_element > k$：左边元素比 K 个大，说明往左边查找。查找 $[L, i-1]$ 中的第 k 大元素即可。由于上述过程并不丢失解，因此该算法正确。

4. 复杂度

最坏情况复杂度：

该算法的复杂度取决于划分函数划分的好坏，与快速排序类似，最坏情况下，每次选择的都是数组中最大或者最小的元素，那么有 $T(n) \leqslant T(n-1) + cn$ 复杂度为 $O(n^{2)}$。

最好情况复杂度：在最好情况下，每次选择为中间的元素，那么有 $T(n) \leqslant T(n/2) + cn$ 复杂度为 $O(n)$。

平均情况复杂度：在平均情况下，我们选择的一般不是最好也不是最差。平均复杂度为 $O(n)$。因为我们这里采用的是随机算法，所以取平均情况时间复杂度。

1.1.2 补充：

数组并没有限定有“特殊的结构”，如果有序，只需要遍历即可。（此处第 k 个最大的元素意指若递增排序，则指从右到左第 k 个元素。）如果我们采用排序算法来给数组加上“结构”，然后遍历，时间复杂度将会达到 $O(nlogn)$。

“若要到达问题的最优解，我们不要把答案要求之外的结构加入到求解过程中。”比如此题，我们如果进行排序，我们是达不到最优解的，因为我们引入了额外的结构：令所有元素都处在排序正确的位置。也即改变给定的 k 之后，我们无需重新运行程序，只需改变输出的 k 即可再次到达正确答案。但是显然，该题并无此要求。给我们的启示就是：是选择而不是排序。不过，若仔细考虑，我们还是能发现它与排序的相同之处：

1. 有序的定义是什么？数组中任一个数之前的数（若有）必定小于等于它，而之后的数（若有）必定大于等于它。

2. 第 k 大元素的定义是什么？数组中第 k 大元素之前的数（若有）必定小于等于它，而之后的数（若有）必定大于等于它。我们并不要求所有数都如此，只要求对于第 k 大元素如此。

这就是二者的区别，就是这种区别，使我们能够抓住问题的本质，并联系到快速排序。因为快速排序的特点是，**每趟排序结束后，总会使一个元素处在确定的位置。并且，相较于其他排序，它是基于元素而不是基于下标划分的，直观来想，它使我们更容易得到第 k 大元素**。此题借用这种思想，但是与快排的不同就是只需递归一边，而且停止条件是找到第 k 大元素即可。

值得注意的是，如果我们假定元素互异，会使论证变得简单，可参照第 4 题。而且，如果我们一直需要选择第 k（k 是变化的）大的元素，显然通过排序构造结构还是明智的。

1.2 完全二叉树的局部最小值

1.2.1 解答：

1. 代码

Algorithm 3 寻找完全二叉树局部最小值

```
FIND-LEAST(root)
1: if root->value<root->left->value AND root-value<root->right->value then
2:     return root->value
3: else if root->left->val < root->val then
4:     return FIND-LEAST(root->left)
5: end if
6: return FIND-LEAST(root->right)
```

用 $C++$ 代码实现如下：

```
int find_least(TreeNode* root)
{
if (root->val < root->left->val && root->val < root->right->val)
        return root->val;
else
    return root->left->val < root->val ?
find_least(root->left) : find_least(root->right);
}
```

2. 归约图

3. 正确性

算法停止有两种情况：

1. v 是个局部最小值，因为它小于它的父结点和孩子结点。

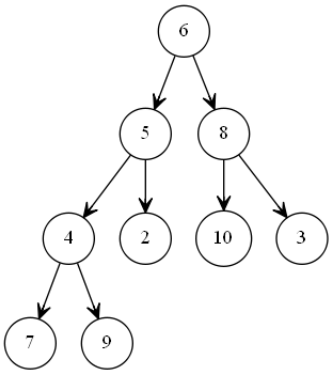
2. v 是个局部最小值，因为它小于它的父结点。无论何种情况，都能得到正确答案，故算法是正确的。

4. 复杂度

显然，最多探测次数（即达到叶节点）与二叉树层数成正比，所以最坏时间复杂度为 $O(logn)$。

1.2.2 补充：

二叉树的许多算法是最能体现分治思想的，在代码中就是用递归完成。关于此题，由于只要求我们返回一个局部最小值即可，所以思路很简单，我们从根结点开始出发，比较其中实数 x_{root} 是否小于左右结点。如果小于，则返回 x_{root} 即可。否则以小于根结点的任一结点作为下一个根结点 $x_{\hat{root}}$，重复以上过程即可，直到叶结点。



我们从 6 出发，与 5,8 比较。来到 5, 与 4,2 比较，来到 4(选 2 也可，来到叶结点，直接返回)。与 7,9 比较，返回 4.

1.3 连续子数组最大和

1.3.1 解答：

1. 代码

Algorithm 4 连续子数组最大和

```
GET-MAX(nums)
return DIVIDE-CONQUER(nums,0,nums.length-1)
```

其中 $DIVIDE-CONQUER$ 定义如下：

Algorithm 5 连续子数组最大和-分治

```
DIVIDE-CONQUER(nums,left,right)
if left==right then
    return (nums[left],nums[left],nums[left],nums[left])
end if
m=(left+right)/2 //取均值
lSub=DIVIDE-CONQUER(nums,left,m)//递归
rSub=DIVIDE-CONQUER(nums,m+1,right)
```

Sum1=lSub.Sum1+rSub.Sum1
lSum=MAX(lSub.lSum,lSub.Sum1+rSub.lSum)//MAX 是取最大值操作
rSum=MAX(rSub.rSum,rSub.Sum1+lSub.rSum)
Sum=MAX(MAX(lSub.Sum,rSub.Sum),left.rSum+right.lSum)
return (lSum,rSum,Sum,Sum1)

用 $C++$ 语言描述如下：

```
class Solution {
public:
struct Status {
    int lSum, rSum, Sum, Sum1;
};
Status Conquer(Status left, Status right) {
    int Sum1 = left.Sum1 + right.Sum1;//区间和直接相加
    int lSum = max(left.lSum, left.Sum1 + right.lSum);
    int rSum = max(right.rSum, right.Sum1 + left.rSum);
    int Sum = max(max(left.Sum, right.Sum), left.rSum + rightlSum);
    return (Status) { lSum, rSum, Sum, Sum1 };
    };
Status get(vector<int>& nums, int left, int right) {
    if (left == right) //基准情况
        return (Status) { nums[left], nums[left], nums[left], nums[left] };
    //分
    int mid = (left + right) >> 1;
    Status lSub = get(nums, left, mid);
    Status rSub = get(nums, mid + 1, right);
    //治
    return Conquer(lSub, rSub);
    }
int maxSubArray(vector<int>& nums) {
    return get(nums, 0, nums.size() - 1).Sum;
}
};
```

2. 归约图

3. 正确性

我们传递的四个信息：$lSum, rSum, Sum1/Sum2, Sum$，我们可以从中选出特定的信息进行组合成下一个合并成的较大数组的信息，到达最后的 Sum 即是答案，它是这样得出的：

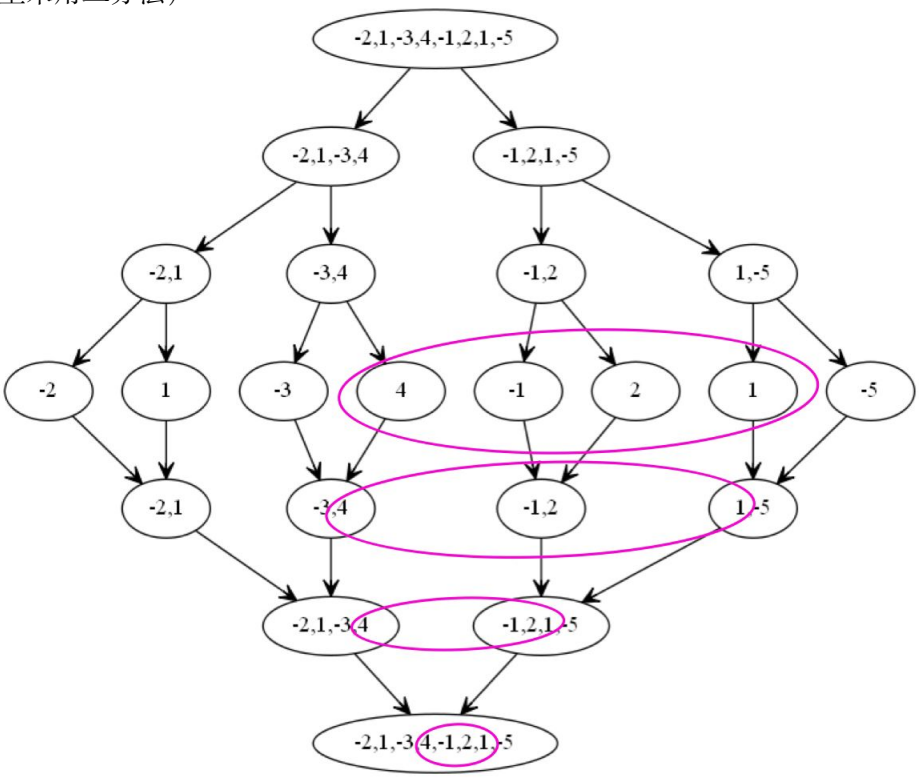
我们可以考虑 Sum 对应的区间是否跨越边界——它可能不跨越，也就是说 Sum 可能是左子数组的 Sum 和右子数组的 Sum 中的一个；它也可能跨越边界，也就是左子数组的 $rSum$ 和右子数组的 $lSum$ 求和。三者取大即可。所以设 Sum 为循环不变量即可，在开始时，Sum 是子数组中的最大值，在每次归并后，仍然保持该属性。归并完成后，仍然成立。

4. 复杂度

假设序列 $nums$ 的长度为 n。我们把递归的过程看作是一颗二叉树的先序遍历，那么这颗二叉树的深度的渐进上界为 $O(logn)$，这里的总时间相当于遍历这颗二叉树的所有节点 (与归并排序唯一不同之处就是这里访问每个结点不是循环，而是一个常量时间 c)，故总时间的渐进上界是 $O(\sum_{i=1}^{logn} 2^{i-1}) = O(n)$。

1.3.2 补充：

这道题比较容易想到的是动态规划：若到达一个数，该数前边的加和是正数，则与之相加；若是负数，则从该数起重新为起点。其间记录最大值即可。那么怎样用分治算法呢？为直观起见，我们举一个简单的例子（这里采用二分法）：



我们不妨先一眼看出答案 $[4,-1,2,1]$（在有正数的例子中，具有最大和的连续子数组一定始于正数，止于正数)。如图中椭圆所示。我们看第四层，我们从中选出了 $[4,-1,2,1]$ 这几个数，但是怎么从彻底分开的基本情况中，将它们联系在一起呢？也就是说，我们组合的时候应该保留什么样的信息，才能从所有的基本情况中选出这几个“特殊的数”？

我们从合并的过程（倒数第三层）看起：4 是 $4,-1$ 中较大的一个数，1 也是 $1,-5$ 中较大的一个数。但是比较划分时两个数的大小显然不够，因为要求“连续”。而至于 $-1,2$ 为什么全包进来，显然也是题中条件” 连续” 所限。

我们观察倒数第二层，4 在 $-2,1,-3,4$ 中有什么特殊的？显然是最大的，但是更准确的应该说应该是以“右端点” 为基准的最大的。为什么强调“右端点”？因为只有这样，才能与右边的形成“连续” 的子数组。而 $-1,2,1$ 在 $-1,2,1,5$ 有什么特殊的？首先它不是 $-1,2,1,5$ 中最大连续子数组和（[2,1] 才是)，所以它加了限制条件，对比前边的限制条件，不难想出这个限制条件就是：以“左端点” 为基准的最大的连续和。为什么强调“左端点”？因为只有这样，才能与左边的形成“连续” 的子数组。

现在我们用左右两个数组 $[l_1,\cdots,r_1]$，$[l_2,\cdots,r_2]$ 合成较大的数组 $[l,\cdots,r]$，应该传递这样的信息：

1. 以 l 为基准的最大值 $lSum$。

2. 以 r 为基准的最大值 $rSum$。

为了用左右两个数组 $[l_1,\cdots,r_1]$, $[l_2,\cdots,r_2]$ 归并成较大数组 $[l,\cdots,r]$ 的时候同样保存以上信息，显然我们还需要：

3. 左右两个数组 $[l_1,\cdots,r_1]$，$[l_2,\cdots,r_2]$ 分别的区间和 $Sum1$、$Sum2$。

这样才能保证跨越子数组组合成更大的连续子数组且能得到 $1,2$ 两条信息。但是，题中的“连续子数组”并没有要求以谁为基准，显然我们还少一条信息：

4. 左右两个数组 $[l_1,\cdots,r_1]$，$[l_2,\cdots,r_2]$ 各个的最大连续和 Sum。（归并到最后的 Sum 即是答案）

我们可以这样得到第四条信息：我们可以考虑 Sum 对应的区间是否跨越 r_1——它可能不跨越，也就是说 Sum 可能是左子数组的 Sum 和右子数组的 Sum 中的一个; 它也可能跨越 r_1，也就是左子数组的 $rSum$ 和右子数组的 $lSum$ 求和。

1.4 在排序数组中查找元素

1.4.1 解答：

1. 代码

Algorithm 6 在排序数组中查找元素

```
SEARCHRANGE(nums,target)
if nums is empty then
    return [-1,-1]
end if
firstPosition = FINDFIRSTPOSITION(nums, target) //如果第 1 次出现的位置都找不到，肯定不存在最后 1 次出现的位置
if firstPosition == -1 then
    return -1, -1
end if
lastPosition = FINDLASTPOSITION(nums, target)
return firstPosition, lastPosition
```

其中 *FINDFIRSTPOSITION* 和 *FINDLASTPOSITION* 分别定义如下：

Algorithm 7 在排序数组中查找元素-第一次出现

```
FINDFIRSTPOSITION(nums,target)
left = 0, right = nums.size() - 1
while left <= right do
    mid = left + (right - left) / 2
    if nums[mid] == target then
        right = mid - 1
    else if nums[mid] < target then
        left = mid + 1
    else
        right = mid - 1
    end if
end while
if left != nums.size() AND nums[left] == target then
    return left
```

```
14: end if
15: return -1
```

Algorithm 8 在排序数组中查找元素-最后一次出现

```
    FINDLASTPOSITION(nums,target)
1:  left = 0, right = nums.size() - 1
2:  while left <= right do
3:    mid = left + (right - left) / 2
4:    if nums[mid] == target then
5:      left = mid + 1
6:    else if nums[mid] < target then
7:      left = mid + 1
8:    else
9:      right = mid - 1
10:   end if
11: end while
12: return right
```

采用 $C++$ 实现如下：（另一种思路）

```cpp
vector<int> searchRange(vector<int>& nums, int target) {
if (nums.empty())return vector<int>{ -1, -1 };
int left = 0, right = nums.size() - 1, mid;
vector<int>ans;
while (left < right)//如果小于目标值，又因为升序排列，所以left变动
{
        mid = left + (right - left) / 2;
        if (nums[mid] < target)
                left = mid + 1;
        else if (nums[mid] >= target)//找到左边第一个等于target的
                right = mid;
}
if (left != nums.size() && nums[left] == target)//对应两种特殊情况
        ans.push_back(left);
left = 0, right = nums.size(), mid;
while (left < right)//如果小于目标值，又因为升序排列，所以left变动
{
    mid = left + (right - left) / 2;
    if (nums[mid] <= target)//找到最后一个大于target的
            left = mid + 1;
    else if (nums[mid] > target)
            right = mid;
}
if (left != 0 && nums[right - 1] == target)
```

```
25          ans.push_back(right - 1);
26  if (!ans.empty())
27          return ans;
28  return vector<int>{ -1, -1 };
29  }
```

2. 归约图

3. 正确性

以 FINDFIRSTPOSITION 为例证明算法的正确性，$FINDLASTPOSITION$ 同理。当 $left \leqslant right$ 时，我们求解 mid 来判断是不是要找的值。如果不是继续以上步骤，直到出现两种情况：

1.$left > right$，此时退出循环，等待进一步判断。

2.$nums[mid] == target$，此时因为可能存在重复元素的原因，不能直接确定是否是第一个元素，令 $right = mid - 1$，继续判断。

退出循环时，我们依据 $left$ 的位置以及对应的值来判断是否找到：

1.$left == nums.length$，显然没有找到。

2. 若不满足 1，且 $nums[left] == target$，则找到了。

可以看到在上述过程中，$left$ 才可能是第一次出现的位置，并且循环的过程中一直保持这个性质，根据循环不变量，若问题有解，则不会丢失。

4. 复杂度

$O(logn)$ ，其中 n 为数组的长度。二分查找的时间复杂度为 $O(logn)$，一共会执行两次，因此总时间复杂度为 $O(logn)$。

1.4.2 补充：

实际上是自己实现类似 $C++lower\_bound$ 和 $upper\_bound$ 函数。$lower\_bound$ 返回第一个等于或大于的迭代器，upper_bound 返回第一个大于的迭代器，若想要取得索引，要减去 begin()。

1.5 凸多边形的三角剖分数目

1.5.1 解答：

1. 代码

Algorithm 9 凸多边形的三角剖分数目

```
GET(n)
if n==3 then
    return 1
end if
for i=3 to n-1 do
    sum=sum+GET(i)*GET(n+2-i)
end for
return (n*sum)/(2*(n-3))
```

但是这样子计算冗余度很高，所以优化如下：

Algorithm 10 凸多边形的三角剖分数目

```
GET(n)
```

```
if n==3 then
  return 1
end if
num[3]=1,sum=0//nums 用来存放中间结果
for i=3 to n-1 do
  if nums[i]==0 then
    nums[i]=GET(i)
  end if
  if nums[n+2-i]==0 then
    nums[n+2-i]=GET(n+2-i)
  end if
  sum+=nums[i]*nums[n+2-i]
end for
return (n*sum) / (2*(n-3))
```

用 $C++$ 代码描述如下：

```
int get(int n)
{
if (n == 3)return 1;
int temp[200] = { 0 };
temp[3] = 1;
int sum = 0;

for (int i = 3; i <= n - 1; i++)
{
        if (temp[i] == 0)
                temp[i] = get(i);
        if (temp[n + 2 - i] == 0)
                temp[n + 2 - i] = get(n + 2 - i);
        sum += temp[i] * temp[n + 2 - i];
}
return (n * sum) / (2 * (n - 3));
}
```

2. 归约图

3. 正确性

我们由“补充”中的推导可以得到递推关系式，据此写程序。由程序可知，$A[i]$ 保存 i 边形的答案，而之后的 $i+1$ 边形需要用到前边的答案。根据归纳法来证明：我们知道 3 边形的条数是正确的，假设中间的结果 ($\leqslant n$) 也是正确的，则由递推式保证最后的答案也是正确的。

4. 复杂度

由代码及归约图容易看出，其递归式为

$$T(n) = T(n-1) + O(n) \tag{1}$$

其右边的 $T(n-1)$ 不好看出，但是若我们换一个角度：求解 $T(n)$ 时只需要计算 $T(n-1)$ 即可，因为在求解 $T(n-1)$ 的过程中会计算出前边所有所需结果。则立马可以得出递推式。至于 $O(n)$ 则是 for 循环中乘法的基本操作次数，实际上为 $n-3$。

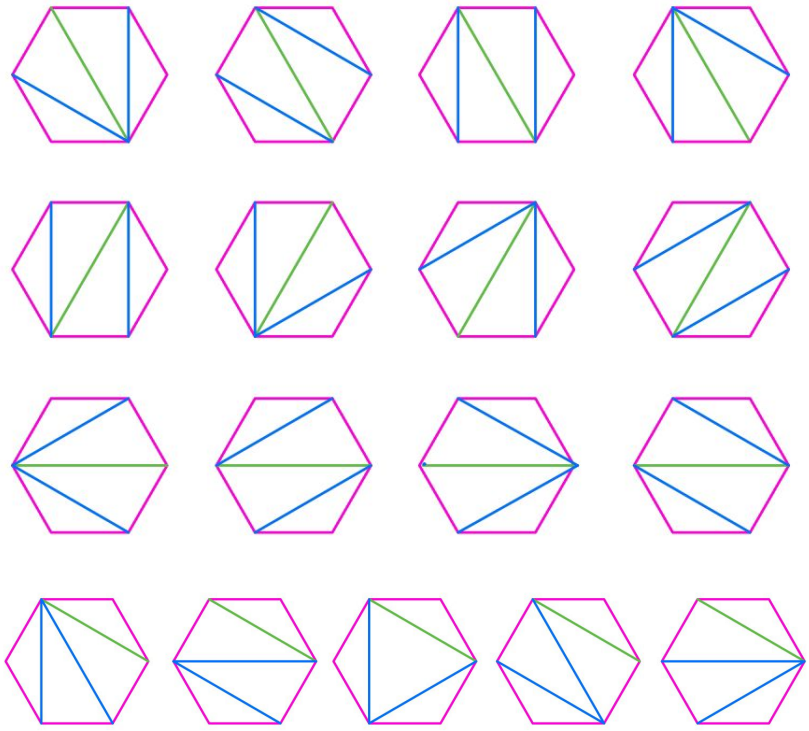
求解如下：

$$\begin{aligned} T(n) &= T(n-1) + c(n-3) \\ &= T(n-2) + c(n-4) + c(n-3) \\ &\cdots \\ &= c(1+2+\cdots+(n-3)) \\ &= c \cdot \frac{n^2-5^n+6}{2} \\ &= O(n^2) \end{aligned}$$

所以复杂度的解为 $O(n^2)$。

1.5.2 补充：

真正读懂一个问题便是将此问题解决了。我们将问题转化为：给定一个凸 n 边形，用 $n-3$ 条不相交对角线将多边形划分为三角形，顺便提一句，对角线条数公式为 $\frac{n(n-3)}{2}$。这里为表达一般性，我们继续给出凸六边形的划分：



如果不考虑重复的部分，我们可以列出递推式：

$$f(n) = \frac{n}{2}[f(3)f(n-1) + f(4)f(n-2) + \cdots + f(n-1)f(3)] \tag{2}$$

其中的 $\frac{n}{2}$ 请读者思考其意义。提示：与顶点数（连成的边）有关。但是很显然，有重复的，那么应该除以多少呢？从题中看出是 $n-3$ 条不相交的对角线，所以，这 $n-3$ 条对角线中都包含了其余 $n-2$ 条

的情况，自然应该除以 $n-3$。可得递推式：

$$f(n)=\frac{\frac{n}{2}[f(3)f(n-1)+f(4)f(n-2)+\cdots+f(n-1)f(3)]}{n-3}\quad n>3 \tag{3}$$

也即：

$$f(n)=\frac{n}{2(n-3)}\sum_{i=3}^{n-1}f(i)f(n+2-i)\quad n>3 \tag{4}$$

其中 $f(n)$ 是 n 边形的划分数，令 $f(3)=1$。不难看出其与**卡特兰数**的关系。

另：本题还有很多其余的推导方法，这里只取一种。

1.6 合并 k 个升序链表

1.6.1 解答：

1. 代码

Algorithm 11 合并升序链表

MERGE(lists,left, right)

```
mid = (left + right) » 1;
return MERGETWOLISTS(MERGE(lists, left, mid), MERGE(lists, mid + 1, right));
```

其中 *MERGETWOLISTS* 定义如下：

Algorithm 12 合并两个升序链表

MERGETWOLISTS(left,right)

```
tail=head,leftPtr=left,rightPtr=right
while leftPtr AND rightPtr do
  if leftPtr->val < rightPtr->val then
    tail->next=leftPtr
    leftPtr=leftPtr->next
  else
    tail->next=rightPtr
    rightPtr=rightPtr->next
  end if
end while
if leftPtr!=nullptr then
  tail=leftPtr
else
  tail=rightPtr
end if
return head->next
```

用 $C++$ 实现如下：

```
ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
if ((!a) || (!b)) return a ? a : b;
ListNode head, * tail = &head, * aPtr = a, * bPtr = b;
while (aPtr && bPtr) {
```

```
        if (aPtr->val < bPtr->val) {
            tail->next = aPtr;
            aPtr = aPtr->next;
        }
        else {
            tail->next = bPtr;
            bPtr = bPtr->next;
        }
        tail = tail->next;
    }
    tail->next = (aPtr ? aPtr : bPtr);
    return head.next;
}

ListNode* merge(vector <ListNode*>& lists, int l, int r) {
    if (l == r) return lists[l];
    if (l > r) return nullptr;
    int mid = (l + r) >> 1;
    return mergeTwoLists(merge(lists, l, mid), merge(lists, mid + 1, r));
}
```

2. 归约图

3. 正确性

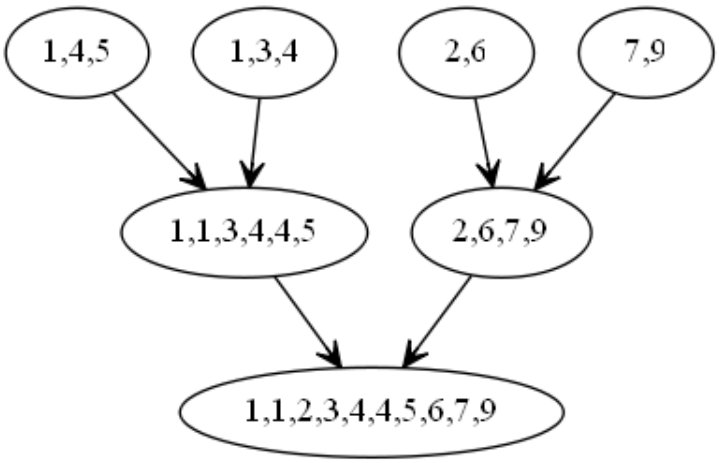
我们设置“归纳不变量”为：每次归并后剩余链表数组中的链表都是按序排好的。刚开始时由题意得知成立，每次归并的过程中，由 $MERGETWOLISTS$ 知中间所生成的链表也保持成立，所以最后归并成一个链表时，仍然成立。正确性得证。

4. 复杂度

由代码及归约图可以看出，这里采用二分法，故树的深度为 $logk$。考虑归并的过程——第一轮归并 $\frac{k}{2}$ 组链表，每一组的时间代价为 $O(2n)$；第二轮归并 $\frac{k}{4}$ 组链表，每一组的时间代价为 $O(4n)\cdots\cdots$ 所以总的时间代价为 $O(\sum_{i=1}^{logk}\frac{k}{2^i}\times 2^i(n)=O(knlogk)$。

1.6.2 补充：

我们会归并两个链表，这里要求归并多个链表，其实本质与两个链表无异。所以这里假设我们已经有了归并两个链表的代码。为直观起见，如下图所示：



可以很明显的看出（由下往上），我们只需要从链表数组中间位置分开然后两边分别调用归并两个链表的代码即可。