北京航空航天大学《算法设计与分析》第一次作业

1 请给出 T(n) 尽可能紧凑的渐进上界并予以说明 (每小题 3 分，共 21 分)

1.1 $\mathbf{T(1) = T(2) = 1}$
$\mathbf{T(n) = T(n-2) + 1 \text{ if } n > 2}$

1.2 $\mathbf{T(1) = 1}$
$\mathbf{T(n) = T(n/2) + n \text{ if } n > 1}$

1.3 $\mathbf{T(1) = 1,\ T(2) = 1}$
$\mathbf{T(n) = T(n/3)} + n^2$ $\mathbf{\text{if } n > 2}$

1.4 $\mathbf{T(1) = 1}$
$\mathbf{T(n) = T(n-1)} + n^2$ $\mathbf{\text{if } n > 1}$

1.5 $\mathbf{T(1) = 1}$
$\mathbf{T(n) = T(n-1)} + 2^n$ $\mathbf{\text{if } n > 1}$

1.6 $\mathbf{T(1) = 1}$
$\mathbf{T(n) = T(n/2)} + logn$ $\mathbf{\text{if } n > 1}$

1.7 $\mathbf{T(1) = 1,\ T(2) = 1}$
$\mathbf{T(n) = 4T(n/3) + n \text{ if } n > 2}$

第1次作业

1 请给出 $T(n)$ 尽可能紧凑的渐进上界并予以说明 (每小题 3 分，共 21 分)

1.

$$T(1) = T(2) = 1$$
$$T(n) = T(n-2) + 1 \quad if \quad n > 2$$

2.

$$T(1) = 1$$
$$T(n) = T(n/2) + n \quad if \quad n > 1$$

3.

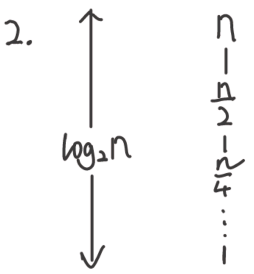
$$T(1) = 1, T(2) = 1$$
$$T(n) = T(n/3) + n^2 \quad if \quad n > 2$$

4.

1. $T(n) = O(n)$

$\because \exists\, C=1, n_0=1$, s.t. $n \geq n_0$ 时 $T(n) \leq n$

2.



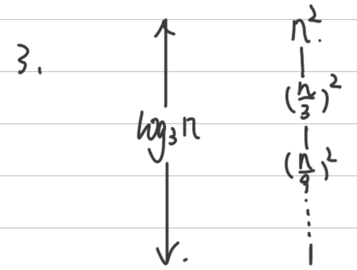
由递归树，得

$T(n) \leq n + \frac{n}{2} + \frac{n}{4} + \cdots$

$= \lim\limits_{t\to\infty} \frac{1\cdot(1-(\frac{1}{2})^t)}{1-\frac{1}{2}} n.$

$= O(n) \quad \therefore T(n) = O(n)$

3.



由递归树，得

$T(n) \leq n^2 + (\frac{n}{3})^2 + (\frac{n}{9})^2 + \cdots$

$= O(n^2)$

$\therefore T(n) = O(n^2)$

Figure 1: 题目 1

4.

$$T(1) = 1$$
$$T(n) = T(n-1) + n^2 \quad if \quad n > 1$$

5.

$$T(1) = 1$$
$$T(n) = T(n-1) + 2^n \quad if \quad n > 1$$

6.

$$T(1) = 1$$
$$T(n) = T(n/2) + \log n \quad if \quad n > 1$$

7.

$$T(1) = 1, T(2) = 1$$
$$T(n) = 4T(n/3) + n \quad if \quad n > 2$$

4. 由递推式可得 $T(n) = 1^2 + 2^2 + \cdots + n^2$

$= \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$

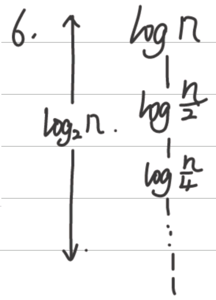
$\therefore T(n)$ 的最紧凑渐进上界为 $O(n^3)$

5. 由递推式：$T(n) = 1 + 2^2 + 2^3 + \cdots + 2^n$

$= 1 + \frac{2^2(1-2^{n-1})}{1-2} = 2^{n+1} - 3. = O(2^n)$

$\therefore T(n) = O(2^n)$

6.



由递归树，$T(n) = \log n + \log \frac{n}{2} + \cdots + 1$

$\leq \log n \cdot \log n + 1$

$= O((\log n)^2)$

7. 由主定理，$\because \log_3 4 > 1 \quad \therefore T(n) = \Theta(n^{\log_3 4})$

$\therefore T(n) = O(n^{\log_3 4})$

Figure 2: 题目 1 续

2 k 路归并问题 (19 分)

现有 k 个有序数组（从小到大排序），每个数组中包含 n 个元素。你的任务是将他们合并成 1 个包含 kn 个元素的有序数组。首先来回忆一下课上讲的归并排序算法，它提供了一种合并有序数组的算法 $Merge$。如果我们有两个有序数组的大小分别为 x 和 y，$Merge$ 算法可以用 $O(x+y)$ 的时间来合并这两个数组。

2.1 如果我们应用 Merge 算法先合并第一个和第二个数组，然后由合并后的数组与第三个合并，再与第四个合并，直到合并完 k 个数组。请分析这种合并策略的时间复杂度（请用关于 k 和 n 的函数表示）。（9 分）

合并第一个和第二个：$O(2n)$

再合并第三个：$O(2n+n)=O(3n)$

...

合并第 k 个：$O((k-1)n+n)=O(kn)$

总时间复杂度：

$$O(2n+3n+\cdots+kn)=O(n(2+3+\cdots+k))=O(n\frac{(k+2)(k-1)}{2})=O(nk^2)$$

2.2 针对本题的任务，请给出一个更高效的算法，并分析它的时间复杂度。（提示：此题若取得满分，所设计算法的时间复杂度应为 O(nk log k)）。（10 分）

2.2.1 思路分析

为了提高效率, 就要想办法减少归并排序的执行次数, 所以我们可以采用分而治之的思想, 每次都将已有的有序数组两两分成一组, 每一组执行归并排序后产生新的数组再重复以上步骤, 直到合并为一个数组。

2.2.2 伪代码实现

Algorithm 1: k 路归并问题优化做法

Input: k 个长度为 n 的数组 $A[]$

Output: 长度为 nk 的数组 Ans，为归并后的数组

```
function main(A):
    while A.length > 1 do
        i = 0, B = [], j = 0
        while i <= A.length − 2 do
            B[j++] = Merge(A[i++], A[i++])
        end
        if i == A.length − 1 then
            B[j++] = A[A.length − 1]
        end
        A = B
    end
    return B
end
```

2.2.3 时间复杂度分析

树的最底层有 k 个节点, 每个节点的长度为 n, 所以最底层的时间复杂度为 $O(\frac{k}{2}\times 2n)=O(nk)$。同理, 倒数第二层的时间复杂度为 $O(\frac{k}{4}\times 4n)=O(nk)$,..., 第二层的时间复杂度为 $O(1\times kn)=O(nk)$，一共

有 $log(k)$ 层。所以总的时间复杂度为 $O(nklogk)$。

3　三余因子和问题 (20 分)

定义整数 i 的“3 余因子”为 i 最大的无法被 3 整除的因子，记作 $md3(i)$，例如 $md3(3) = 1, md3(18) = 2, md3(4) = 4$。请你设计一个高效算法，计算一个正整数区间 [A, B],(0 < A < B) 内所有数的“3 余因子”之和，即 $\sum_{i=A}^{B} md3(i)$，并分析该算法的时间复杂度。例如，区间 [3, 6] 的计算结果为 $1 + 4 + 5 + 2 = 12$。

3.1　思路分析

首先对“3 余因子”进行分析，可以得到以下结论：

$$\begin{cases} \text{对于非 3 的倍数，其 3 余因子为其本身} \\ \text{对于 3 的倍数，其 3 余因子为其最大的非 3 的倍数的因子} \end{cases}$$

因此，我们只要对给定范围内 3 的倍数和非 3 的倍数分别讨论，然后用对应的方式求出其“3 余因子”并求和即可。

对于求和，我们可以考虑采用先整体后部分的思想，就是先求出范围内所有整数之和，然后减去所有是 3 的倍数的整数之和，之后再加上它们除以三后的整数之和，如此循环下去，直到最终所有三的倍数都不可再除为止。

3.2　伪代码实现

Algorithm 2: 三余因子和问题

```
Input: 正整数区间 [Left, Right]
Output: 区间内所有数的“3 余因子”之和 sum
function threeSum(Left, Right):
    sum = 0
    if Left > Right then
        return 0
    end
    if Left%3 == 0 then
        nl = Left/3
    end
    else
        nl = Left/3 + 1
    end
    nr = Right/3
    sum = (Left+Right)*(Right−Left+1)/2−(nl+nr)*(nr−nl+1)/2*3+threeSum(nl, nr)
    return sum
end
function main:
    sum = threeSum(Left, Right)
    return sum
end
```

3.3 时间复杂度分析

对于给定的区间，算出三余因子之和只需要进行一次递归即可，而递归每层的时间复杂度为 $O(1)$，递归层数为 $logn$，因此总的时间复杂度为 $O(logn)$。

4 填数字问题 (20 分)

给定一个长度为 n 的数组 A[1..n]，初始时数组中所有元素的值均为 0，现对其进行 n 次操作。第 i 次操作可分为两个步骤:

1. 先选出 A 数组长度最长且连续为 0 的区间，如果有多个这样的区间，则选择最左端的区间，记本次选定的闭区间为 [l, r]；

2. 对于闭区间 [l, r]，将 A[$\lfloor \frac{l+r}{2} \rfloor$] 赋值为 i，其中 $\lfloor x \rfloor$ 表示对数 x 做向下取整。

例如 n = 6 的情形，初始时数组为 A = [0, 0, 0, 0, 0, 0]。

第一次操作为选择区间 [1, 6]，赋值后为 A = [0, 0, 1, 0, 0, 0]；

第二次操作为选择区间 [4, 6]，赋值后为 A = [0, 0, 1, 0, 2, 0]；

第三次操作为选择区间 [1, 2]，赋值后为 A = [3, 0, 1, 0, 2, 0]；

第四次操作为选择区间 [2, 2]，赋值后为 A = [3, 4, 1, 0, 2, 0]；

第五次操作为选择区间 [4, 4]，赋值后为 A = [3, 4, 1, 5, 2, 0]；

第六次操作为选择区间 [6, 6]，赋值后为 A = [3, 4, 1, 5, 2, 6]，为所求。

请设计一个高效的算法求出 n 次操作后的数组，并分析其时间复杂度。

4.1 思路分析

如果采用每次寻找最长 0 区间的方法，时间复杂度会达到 $O(n^2)$。为了降低时间复杂度，我们可以考虑使用分而治之的思想，按照区间中点. 将区间建立成一棵线段树。每次操作时，从根节点开始，在线段树中递归搜索最长的连续为 0 的区间，找到该区间后，更新其中点的值，然后递归更新左右子树的信息，重复上述提作直到构建完结果数组。

本题更详细分析见本人博客：https://1314189.xyz/posts/1df69f66.html

4.2 伪代码实现

Algorithm 3: 填数字问题

Input: 数组 $A[1..n]$

Output: 填充后的数组 $A[1..n]$

Class TreeNode:

function $initTreeNode(self, left, right)$**:**

  $self.left = left$

  $self.right = right$

  $self.length = right - left + 1$

  $self.mid = \lfloor \frac{left+right}{2} \rfloor$

  $self.value = 0$

  $self.left\_node = None$

  $self.right\_node = None$

end

Algorithm 4: 填数字问题（续）

```
function buildTree(left, right):
    mid = ⌊(left+right)/2⌋
    if left == right then
        return TreeNode(left, right)
    end
    left_node = buildTree(left, mid)
    right_node = buildTree(mid + 1, right)
    return TreeNode(left, right)
end
function updateTree(node, idx, value):
    if node.left_node.length == node.length then
        rs = updateTree(node.left_node, idx, value)
    end
    else if node.right_node.length == node.length then
        rs = updateTree(node.right_node, idx, value)
    end
    else
        rs = node.mid //找到最长的连续为 0 的区间，即目前的 node
        updateValue(node, rs, value)
    end
    node.length = max(node.left_node.length, node.right_node.length)
    return rs
end
function updateValue(node, idx, value):
    //更新已找到的最长连续为 0 序列中点的值，同时递归更新后续各节点的长度
    if node.left == node.right then
        node.value = value
        node.length = 0
        return
    end
    else if idx <= node.mid then
        updateValue(node.left_node, idx, value)
    end
    else
        updateValue(node.right_node, idx, value)
    end
    node.length = max(node.left_node.length, node.right_node.length)
end
function fillArray(n):
    root = buildTree(1, n)
    result = []
    for i = 1 → n do
        max_length = root.length
        result_idx = root.mid
        rs = updateTree(root, result_idx, i)
        result[rs − 1] = i
    end
    return result
end
```

4.3 时间复杂度分析

由于每次查找最长连续为 0 区间耗时为 $logn$，建树的时间复杂度为 $O(nlogn)$，因此总的时间复杂度为 $O(nlogn)$。

5 数字消失问题 (20 分)

给定一长度为 n 的数组 A[1..n]，其包含 [0, n] 闭区间内除某一特定数 (记做消失的数) 以外的所有数字（例如 n = 3 时，A = [1, 3, 0]，则消失的数是 2）。这里假定 $n = 2^k - 1$。

5.1 请设计一个尽可能高效的算法找到消失的数，并分析其时间复杂度。(8 分)

5.1.1 思路分析

由于给定数组 A 无序, 不便于寻找消失的数, 因此我考虑按照一定的方式将 A 有序化。直接对 A 排序显然比较耗时, 但如果我们通过遍历一遍 A, 就能将 A 中已有的数都有序地表示出来是可行的, 只需要设定一个 B 数组, 其索引就对应 [0,n] 中的数, 而其值就代表这个数是否存在。这样一来, 遍历一遍 A 就可完成 B 数组的构建, 再遍历一遍 B 就能找到消失的数。

5.1.2 伪代码实现

Algorithm 5: 数字消失问题（1）

```
Input: 数组 A[1..n]
Output: 消失的数
function findMissingNumber(A):
    B[n + 1] = 0
    for i = 0 → n − 1 do
        B[A[i]] = 1
    end
    for i = 0 → n do
        if B[i] == 0 then
            return i
        end
    end
end
```

5.1.3 时间复杂度分析

由于遍历 A 和 B 的时间复杂度都是 $O(n)$, 因此总的时间复杂度为 $O(n)$。

5.2 若假定数组 A 用 k 位二进制方式存储（例如 k = 2，A = [01, 11, 00] 则消失的数是 10），且不可以直接访存（即不可以直接通过数组的下标访问数组的内容）。目前唯一可以使用的操作是 bit-lookup(i, j)，其作用是用一个单位时间去查询 A[i] 的第 j 个二进制位。请利用此操作设计一个尽可能高效的算法找到消失的数，并分析其时间复杂度。(12 分)

5.2.1 思路分析

由于每次只能读取一位, 我们就按位进行分析, 对于数组 A 而言, 所有数的相同位上 0 的个数和 1 的个数之和仅差 1, 而少的那个就是消失的数在这一位上的值, 通过该方法便能求出消失的数每一位的值.

5.2.2 伪代码实现

Algorithm 6: 数字消失问题（2）

```
Input: 数组 A[1..n]
Output: 消失的数
function findMissingNumber(A):
    dis = 0
    for i = 0 → k − 1 do
        count_0 = 0
        count_1 = 0
        for j = 0 → n − 1 do
            if bit − lookup(A[j], i) == 0 then
                count_0 + +
            end
            else
                count_1 + +
            end
        end
        if count_0 < count_1 then
            dis+ = 1 << i
        end
        else
            dis+ = 0
        end
    end
    return dis
end
```

5.2.3 时间复杂度分析

由于遍历 A 的时间复杂度为 $O(n)$, 而对于每一位的遍历时间复杂度为 $O(k)$, 因此总的时间复杂度为 $O(nk)$。