计算机算法设计与分析

083500M01001H

Chap 3 课程作业解答

2022 年 9 月 20 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

讨论归并排序算法 **MergeSort** 的空间复杂性.

Solution:

归并排序的递归调用过程需要 $O(h)$ 的栈空间 (h 为递归树的高度), 而整个递归树的高度 (即递归调用的最深层数) 为 $\log n$, 在合并过程中也需要额外 $O(n)$ 空间的 temp 数组 (而快速排序却不需要). 故归并排序和快速排序的空间复杂度分别为 $O(n + \log n) = O(n)$, $O(\log n)$.

Problem 2

改进插入排序算法 (第三章 ppt No.6), 在插入元素 $a[i]$ 时使用二分查找代替顺序查找, 将这个算法记做 **BinarySort**, 估计算法在最坏情况下的时间复杂度.

Solution:

先写出 **BinarySort** 算法的伪代码, 如下所示:

Algorithm 1 二分插入排序 **BinarySort** 算法

Input: 长度为 n 的数组 $A[0, \cdots, n-1]$
Output: 按递增次序排序的 A

```
1:  for i := 1 to n − 1 do
2:      int temp = A[i];                  ▷ 其实第 3 行也可这样: int low = upperbound(A, 0, i − 1, temp);
3:      int low = upper_bound(A.begin(), A.begin() + i, temp) − A.begin();     ▷ 源于 C++ 的 STL 标准库
4:      if low ! = i then                 ▷ 若 low=i, 说明 A[0, ··· , i − 1] 中没有 temp 的插入位置
5:          for j from i − 1 by −1 to low do
6:              A[j + 1] := A[j];
7:          end for
8:          A[low] = temp;
9:      end if
10: end for
11: end {BinarySort};
```

接下来分析最坏情况下的时间复杂度:

证明. 最坏情况显然是逆序的数组. 不管是用二分查找还是顺序查找, 都只能在查找位置上节约时间, 但是算法的**关键操作**是数组遍历和元素后移, 而需要遍历 $1 + 2 + \cdots + (n-1) = \frac{1}{2}n(n-1) = \Theta(n^2)$. □

Problem 3

设 A 是 n 个非 0 实数构成的数组, 设计一个算法重新排列数组中的数, 使得负数都排在正数前面, 要求算法复杂度为 $O(n)$.

Solution: 方法 1 (时空复杂度分别为 $O(n)$, $O(1)$): 直接调用快速排序中的 partition 函数并令 pivot=0 即可.

方法 2 (时空复杂度分别为 $O(n)$, $O(1)$): 具体见算法2.

Algorithm 2 三色国旗问题的 **ThreeColor** 算法

Input: n 个实数构成的数组 $A[0,\cdots,n-1]$
Output: 负数排在正数前面且 0 排在中间的数组 $A[0,\cdots,n-1]$

int pivot = 0;
int lt = -1, i = 0, gt = n;
while i < gt **do**
  if $A[i]$ == pivot **then**
    i++;
  else if $A[i]$ > pivot **then**
    swap($A[i]$, $A[\mathrm{gt}-1]$), gt- -;
  else if $A[i]$ < pivot **then**
    swap($A[\mathrm{lt}+1]$, $A[i]$), lt++, i++;
  end if
end while
end {ThreeColor}

Problem 4

Hanoi 塔问题: 图中有 A, B, C 三根柱子, 在 A 柱上放着 n 个圆盘, 其中小圆盘放在大圆盘的上边. 从 A 柱将这些圆盘移到 C 柱上去, 在移动和放置时允许使用 B 柱, 但不能把大盘放到小盘的下面. 设计算法解决此问题, 分析算法复杂度.

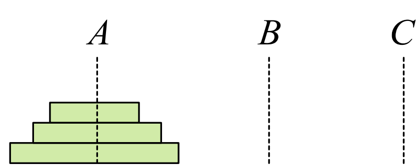

图 1: 汉诺塔问题

Solution:

该问题即为著名的汉诺塔问题, 递归式的求解算法描述为: 先将 A 上面的 $n-1$ 个盘子移到 B, 再将 A 中最下边的盘子移动到 C, 再将 B 中的 $n-1$ 个盘子移动到 C 上即可. 伪码描述为算法3:

Algorithm 3 汉诺塔问题的递归算法 **Hanoi**(A, C, n)

Input: n 个盘子从上往下、从小到大放在 A 柱

Output: 将 A 柱的圆盘移到 C 柱上

```
if n == 1 then
    move (A, C);
else
    Hanoi(A, B, n − 1);
    move (A, C);
    Hanoi(B, C, n − 1);
end if
end {Hanoi}
```

易知 $T(1) = 1$, 根据上述伪码可知时间复杂度有递推方程

$$T(n) = 2T(n-1) + 1 \tag{1}$$

于是通过递推得到

$$\begin{aligned} T(n) &= 2T(n-1) + 1 = 2(2T(n-2) + 1) + 1 \\ &= 2^2 T(n-2) + 1 + 2^1 \\ &= 2^3 T(n-3) + 1 + 2^1 + 2^2 \\ &= 2^{n-1} T(1) + 1 + 2 + \cdots + 2^{n-2} \\ &= 2^n - 1 \end{aligned}$$

于是 $T(n) = \Theta(2^n)$. 而且可以证明的是, 汉诺塔问题不存在多项式时间算法, 因此是一个难解的问题.

Problem 5

给定含有 n 个不同数的数组 $L=\{x_1,x_2,\cdots,x_n\}$, 若 L 中存在 x_i, 使得 $x_1<x_2<\cdots<x_{i-1}<x_i>x_{i+1}>\cdots>x_n$, 则称 L 是单峰的, 并称 x_i 是 L 的峰顶. 假设 L 是单峰的, 设计一个优于 $O(n)$ 的算法找到 L 的峰顶.

Solution:

算法思路描述: 对区间 $[0,n-1]$ 进行二分, 不妨设中点为 mid= $\lfloor (n-1)/2 \rfloor$.
观察中点的左右邻点:

- **case1:** 若 $L\,[\text{mid}-1] < L\,[\text{mid}] < L\,[\text{mid}+1]$, 则显然峰顶在右半区间 $[\text{mid}+1, n-1]$, 在该区间继续二分搜索即可;
- **case2:** 若 $L\,[\text{mid}-1] > L\,[\text{mid}] > L\,[\text{mid}+1]$, 则显然峰顶在左半区间 $[0, \text{mid}-1]$, 在该区间继续二分搜索即可;
- **case2:** 若 $L\,[\text{mid}-1] < L\,[\text{mid}] > L\,[\text{mid}+1]$, 则显然峰顶就是 $L\,[\text{mid}]$, 至此搜索完毕.

对应的算法伪代码见如下:

Algorithm 4 单峰数组的二分搜索算法 **peakIndex**(L)

Input: n 个不同数的数组 $L[0,n-1]$ 且 L 为单峰数组
Output: L 的峰顶

if n == 3 **then** ▷ 因为已知 L 为单峰数组, 因此 $n \geq 3$
  return $L[1]$; ▷ 若 $n=3$, 则显然 $L[1]$ 为峰顶
else
  int low = 0, high = $n-1$;
  while low <= high **do**
    int mid = (low + high) >> 1;
    if $L\,[\text{mid}-1] < L\,[\text{mid}] > L\,[\text{mid}+1]$ **then**
      return $L\,[\text{mid}]$;
    else if $L\,[\text{mid}-1] < L\,[\text{mid}] < L\,[\text{mid}+1]$ **then**
      low = mid + 1;
    else
      high = mid − 1;
    end if
  end while
end if
end {peakIndex}

现在来分析算法的时间复杂度 $T(n)$: 因为每一次二分都只需要做两次 (常数次) 比较, 所以 $T(n)$ 的递归方程为 $T\,(n) = T\,(n/2) + O\,(1)$, 根据主定理 $(a=1, b=2, d=0)$ 可解得 $T\,(n) = O\,(\log n)$.

Problem 6

设 A 是 n 个不同元素组成且排好序的数组, 给定数 L 和 U, $L < U$, 设计一个优于 $O(n)$ 的算法, 找到 A 中满足 $L < x < U$ 的所有数 x.

Solution: 重新写!!!!

算法思路描述: 我们需要分类讨论:

- 当 $L >= A[n-1]$ 或 $U <= A[0]$ 时, 显然数集 $x = \emptyset$;
- 当 $L < A[0] < U < A[n-1]$ 时, 用下述的**lowerbound 算法**找到数组 A 中第一个大于等于 U 的元素索引 q, 则 $x = \{A[0], A[1], \cdots, A[q-1]\}$;
- 当 $L = A[0] < U < A[n-1]$ 时, 则 $x = \{A[1], A[2], \cdots, A[q-1]\}$;
- 当 $A[0] < L < U < A[n-1]$ 时, 则先用 **Problem 5** 的 **upperbound 算法**找到数组 A 中第一个大于 L 的元素索引 p, 再用**lowerbound 算法**找到数组 A 中第一个大于等于 U 的元素索引 q, 这样就有

$$x = \{A[p], A[p+1], \cdots, A[q-1]\}$$

- 当 $A[0] < L < U = A[n-1]$ 时, 则用 **upperbound 算法**找到数组 A 中第一个大于 L 的元素索引 p, 于是 $x = \{A[p], A[p+1], \cdots, A[n-2]\}$;
- 当 $A[0] < L < A[n-1] < U$ 时, 则用 **upperbound 算法**找到数组 A 中第一个大于 L 的元素索引 p, 则 $x = \{A[p], A[p+1], \cdots, A[n-1]\}$;

Algorithm 5 二分查找 **lowerbound** 算法

Input: 长度为 n 的升序数组 $A[0, \cdots, n-1]$, 目标元素 target

Output: 第一个大于等于 target 的元素下标

```
int low = 0, high = n − 1;
while low <= high do
    int mid = (low + high) >> 1;
    if A[mid] < target then
        low = mid + 1;
    else
        high = mid − 1;
    end if
end while
return low;                                  ▷ 返回所要求的下标
end {lowerbound}
```

此题主要在于分类讨论和第 4 种情况的求解, 其对应的 C++ 程序非常简单 (直接用一下 STL 的 lower_bound 和 upper_bound 二分查找函数即可), 故此处就不再罗列了. 而本文所构造的算法主要用到了两个二分查找的函数, 显然该算法的时间复杂度为 $O(\log n)$, 是优于 $O(n)$ 的.

Problem 7

设 $A=\{a_1,a_2,\cdots,a_n\}$, $B=\{b_1,b_2,\cdots,b_m\}$ 是整数集合, 其中 $m=O(\log n)$, 设计一个优于 $O(nm)$ 的算法找出集合 $C=A\cap B$.

Solution:

方法 2 (排序 + 二分查找): 算法思想描述: 由于数组 B 比较短, 所以先对其进行排序, 然后遍历数组 A 的元素, 在排序后的 B 中使用二分查找来检索该元素, 若找到则放入 C 中. 算法伪码描述如下:

Algorithm 6 数组交集算法 **Intersection**

Input: 数组 $A[0,\cdots,n-1]$, $B[0,\cdots,m-1]$
Output: 数组 C, 其中 $C=A\cap B$

vector<int> C;
sort(B.begin(), B.end()); ▷ 对数组 B 进行原地排序
for $i=0$; $i<n$; i++ **do**
  bool flag = **binary_search**(B.begin(), B.end(), $A[i]$);
  if flag == `true` **then**
    C.**push_back**($A[i]$);
  end if
end for
return C;
end {Intersection} ▷ 算法的 C++ 代码跟伪代码非常相似, 就不列出了

现在来分析一下此方法的时空复杂度: 对 B 排序需要 $O(m\log m)$ 的时间, 遍历 + 二分查找需要消耗 $O(n\times\log m)$ 的时间, 所以总的时间复杂度为 $O(m\log m)+O(n\log m)=O((m+n)\log m)=O(n\log m)=O(n\log\log n)$; 而数组 B 排序所用到的栈空间为 $O(\log m)$, 对 B 二分查找所需的栈空间为 $O(\log m)$, 所以算法的空间复杂度为 $O(\log m)=O(\log\log n)$.

Problem 8

设 S 是 n 个不等的正整数的集合, n 为偶数, 给出一个算法将 S 划分为子集 S_1 和 S_2, 使得 $|S_1| = |S_2|$ 且 $\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$ 达到最大, 即两个子集元素之和的差达到最大 (要求时间复杂度 $T(n) = O(n)$).

Solution:

算法思想描述: 先利用 **PartSelect 算法**选取数组 S 的第 $n/2+1$ 小的元素, 并将该元素作为 pivot 并利用 **Partition 算法**来对数组 S 进行**一次划分**, 低区元素全部进入 S_2, 高区元素和 pivot 都进入到 S_1(由于 n 为偶数, 所以能够保证 $|S_1| = |S_2|$), 这样就能够确保 $\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$ 达到最大. 算法伪码见如下:

Algorithm 7 最大化子集和差算法 **MaxSubtract**

Input: 数组 $S[0,\cdots,n-1]$ ▷ n 为偶数, S 的元素彼此互异都为正整数

Output: $\max_{|S_1|=|S_2|}\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$

```
vector<int> S1(n/2), S2(n/2);
int pivot = PartSelect (S, 0, n − 1, n/2 + 1);    ▷ 求数组 S 第 n/2 + 1 小的元素, 可以认为是" 中位数"
int low = 0, high = n − 1;    ▷ 对撞双指针做一次划分
while low < high do
    while low < high && S[high] >= pivot do
        high--;
    end while
    swap(S[low], S[high]);
    while low < high && S[low] <= pivot do
        low++;
    end while
    swap(S[low], S[high]);
end while
int loc = low;    ▷ 此时的 loc 即为 pivot 所处的最终下标
copy(S.begin(), S.begin() + loc, S2.begin());    ▷ 低区进入 S2
copy(S.begin() + loc, S.begin() + (n − loc), S1.begin());    ▷ 高区和 pivot 进入 S1
return S1, S2;    ▷ 注意到 loc 其实就是 n/2, 因此 n − loc 即为 n/2
return accumulate(S1.begin(), S1.end(), 0) − accumulate(S2.begin(), S2.end(), 0);
end {MaxSubtract}
```

现在来分析一下算法的时间复杂度: 调用 **PartSelect 算法**最坏需要 $O(n)$ 的时间, **Partition 算法** (核心是对撞双指针) 需要 $O(n)$ 的时间, 所以总的时间复杂度为 $T(n) = O(n) + O(n) = O(n)$.

Problem 9

考虑第三章 PPT NO.17 **Select**(A, k) 算法:

(1). 如果初始元素分组 $r = 3$, 算法的时间复杂度如何? (2). 如果初始元素分组 $r = 7$, 算法的时间复杂度如何?

Solution:

(1). 若 $r = 3$, 既可以认为子问题的规模为 $2n/3$. 求中位数的中位数所递归调用的规模为 $n/3$, 一趟快排和插入排序的所需时间为 cn. 综上, 时间复杂度的递推方程为

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \tag{2}$$

画出 $T(n)$ 的递归树, 见如下图2:

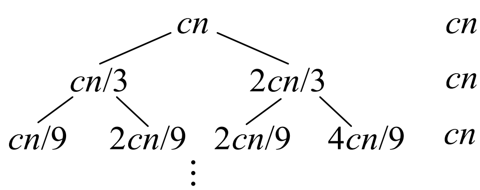

图 2: 递归树-1

而递归树的深度为 $\log n$, 而每一层的操作都是 cn, 所以时间复杂度 $T(n) = O(cn \log n) = O(n \log n)$;
(2). 若 $r = 7$, 即可以认为子问题的规模为 $5n/7$. 求中位数的中位数所递归调用的规模为 $n/7$, 一趟快排和插入排序的所需时间为 cn. 综上, 时间复杂度的递推方程为

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + cn \tag{3}$$

画出 $T(n)$ 的递归树, 见如下图3:

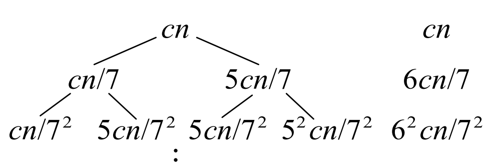

图 3: 递归树-2

故总的时间复杂度为:

$$T(n) = cn\left(1 + \frac{6}{7} + \left(\frac{6}{7}\right)^2 + \cdots\right) \le cn \cdot \frac{1}{1 - \frac{6}{7}} = 7cn = O(n) \tag{4}$$

Problem 10

对玻璃瓶做强度试验, 设地面高度为 0, 从 0 向上有 n 个高度, 记为 $1, 2, \cdots, n$, 其中任何两个高度之间的距离都相等. 如果一个玻璃瓶从高度 i 落到地上没有摔碎, 但从高度 $i+1$ 落到地上摔碎了, 那么就将玻璃瓶的强度记为 i.

(1). 假设每种玻璃瓶只有 1 个测试样品, 设计算法来测试出每种玻璃瓶的强度. 以测试次数作为算法的时间复杂度, 估计算法的复杂度;

(2). 假设每种玻璃瓶有足够多的相同的测试样品, 设计算法使用最少的测试次数来完成测试;

(3). 假设每种玻璃瓶只有 2 个相同的测试样品, 设计次数尽可能少的算法完成测试.

Solution:

(1). 顺序从下到上测试, 一次一个高度, 最坏情况下时间复杂度为 $T(n) = O(n)$;

(2). 因为高度越高, 玻璃瓶越容易碎, 其实可以理解为" 升序数组". 因此我们可以考虑用二分法: 先在高度 $n/2$ 测试玻璃瓶, 如果摔碎了, 则玻璃瓶的强度位于 $[1, n/2-1)$(在该区间继续二分搜索即可); 若没摔碎, 则玻璃瓶的强度位于 $[n/2+1, n]$(在该区间继续二分搜索即可). 显然, 该二分搜索的时间复杂度为 $T(n) = O(\log n)$;

(3). 不失一般性, 不妨设 $\sqrt{n}$ 为整数, 则可以将 $1, 2, 3, \cdots, n$ 这些 n 个高度分成 $\sqrt{n}$ 组[1]. 那么第 j 组 $(j = 1, 2, \cdots, \sqrt{n})$ 所含有的高度有

$$(j-1)\sqrt{n}+1, (j-1)\sqrt{n}+2, \cdots, (j-1)\sqrt{n}+\sqrt{n}, \quad j = 1, 2, \cdots, \sqrt{n} \tag{5}$$

先拿第一个瓶子测试: 从下往上, 按照每组的最大高度 (即 $j\sqrt{n}, j = 1, 2, \cdots, \sqrt{n}$) 进行测试. 如果前 $j-1$ 组的测试中瓶子都没有碎, 而在第 j 组的测试中碎了, 则强度显然位于第 j 组的 $\sqrt{n}$ 个高度中. 于是, 至多经过 $\sqrt{n}$ 此测试, 待检查的高度范围就缩减到原来的 $\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}}$ 倍;

再拿第二个瓶子测试: 在第 j 组的 $\sqrt{n}$ 个高度中, 从下往上测试玻璃瓶的强度, 至多经过 $\sqrt{n}$ 次测试, 就可以得到玻璃瓶的强度.

现在来分析算法的时间复杂度: 显然第一个瓶子测验至多需要耗时 $O(\sqrt{n})$, 第二个瓶子测试也至多需要耗时 $O(\sqrt{n})$, 于是总的算法时间复杂度为

$$T(n) = O(\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n}) \tag{6}$$

[1]如果 $\sqrt{n}$ 不是整数, 则取 $\lfloor \sqrt{n} \rfloor$ 个整组, 剩下的单独成一组

Problem 11

1. 使用主定理求解以下递归方程:

(1). $\begin{cases} T(n) = 9T(n/3) + n \\ T(1) = 1 \end{cases}$; (2). $\begin{cases} T(n) = 5T(n/2) + (n\log n)^2 \\ T(1) = 1 \end{cases}$; (3). $\begin{cases} T(n) = 2T(n/2) + n^2\log n \\ T(1) = 1 \end{cases}$

Solution:

(1). 易知 $a = 9, b = 3, d = 1, f(n) = n$, 由于 $f(n) = n = O\left(n^{2-\epsilon}\right)$, 故根据主定理可知: $T(n) = \Theta(n^2)$;

(2). 易知 $a = 5, b = 2, f(n) = n^2\log^2 n = O(n^{\log_2 5-\epsilon})$, 故根据主定理可知 $T(n) = \Theta\left(n^{\log_2 5}\right)$;

(3). 易知 $a = 2, b = 2, f(n) = n^2\log n = \Omega\left(n^{1+\epsilon}\right)$, 而且

$$af(n/b) = 2(n/2)^2\log(n/2) = n^2/2(\log n - 1) \le 0.5n^2\log n \quad (c = 1/2 < 1)$$

故根据主定理可知 $T(n) = \Theta(f(n)) = \Theta(n^2\log n)$.

2. 使用递归树求解: $\begin{cases} T(n) = T(n/2) + T(n/4) + cn \\ T(1) = 1 \end{cases}$;

Solution:

递归树见如下图4:

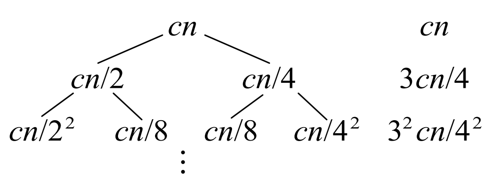

图 4: 递归树

故总的时间复杂度为:

$$T(n) = cn\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \cdots\right) \le cn\cdot\frac{1}{1-\frac{3}{4}} = 4cn = O(n) \tag{7}$$

3. 使用迭代递归法求解: (1). $\begin{cases} T(n) = T(n-1) + \log 3^n \\ T(1) = 1 \end{cases}$; (2). $\begin{cases} T(n) = T(n-1) + 1/n \\ T(1) = 1 \end{cases}$.

Solution:

(1). 易知

$$T(n) = T(n-1) + \log 3^n = T(n-2) + \log 3^{n-1} + \log 3^n \tag{8}$$
$$\cdots = T(1) + \log 3^2 + \log 3^3 + \cdots + \log 3^n \tag{9}$$
$$= 1 + \log\left(3^{2+3+\cdots+n}\right) = 1 + \log\left(3^{(n+2)(n-1)/2}\right) = \Theta\left(n^2\right) \tag{10}$$

(2). 易知

$$T(n) = T(n-1) + \frac{1}{n} = T(n-2) + \frac{1}{n-1} + \frac{1}{n} \tag{11}$$
$$\cdots = T(1) + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n} = \sum_{i=1}^{n}\frac{1}{i} = \Theta(\gamma + \log n) = \Theta(\log n) \tag{12}$$

注意, 其中我们用到了 γ 常数的数学结论: $\gamma = \lim\limits_{n\to+\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$.