University of Chinese Academy of Sciences

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

编写程序实现归并排序算法 **MergeSort** 和快速排序算法 **QuickSort**;

Solution: 以下是手撕归并排序算法 **MergeSort** 的 C++ 程序.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
void Merge(vector<int>& nums, int low, int mid, int high) {
    int i = low, j = mid + 1, k = 0;
    vector<int> temp(high - low + 1);
    while(i <= mid && j <= high) {
        if(nums[i] <= nums[j])
            temp[k++] = nums[i++];
        else
            temp[k++] = nums[j++];
    }
    while(i <= mid) temp[k++] = nums[i++];
    while(j <= high) temp[k++] = nums[j++];
    for(int i = low; i <= high; i++) {
        nums[i] = temp[i - low];
    }
}
void MergeSort(vector<int>& nums, int low, int high) {
    if(low < high) {
        int mid = (high + low) >> 1;
        MergeSort(nums, low, mid);
        MergeSort(nums, mid + 1, high);
        Merge(nums, low, mid, high);
    }
}
int main() {
    int n;
    cin >> n;//输入数组的长度
    vector<int> a(n);//定义数组
    for (int i = 0; i < n; i++) {//对数组初始化
        a[i]= -1 * (n + 1) + rand()%(2 * n + 2); //在 [-n-1,n+1] 随机产生数组
        cout << a[i] << " ";  //打印该数组
    }
    printf("\n");
    clock_t startTime = clock();//计时开始
    MergeSort(a, 0, n - 1); //手撕归并排序, 20000000 的数据量耗时 6.016s
    clock_t endTime = clock();//计时结束
    for(int i = 0; i < n; i++) {
        cout << a[i] << " "; //输出排序后的数组
    }
    printf("\n");
    cout << " 归并排序算法的运行时间为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

Solution: 以下是手撕快速排序算法 **QuickSort** 的 C++ 程序.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int partition(vector<int>& nums, int low, int high) {
    int index = low + rand()%(high - low + 1);
    swap(nums[low], nums[index]);
    int pivot = nums[low];
    while (low < high) {
        while (low < high && nums[high] >= pivot) high--;
        swap(nums[low], nums[high]);
        while (low < high && nums[low] <= pivot) low++;
        swap(nums[low], nums[high]);
    }
    return low;
}
void QuickSort(vector<int>& nums, int low, int high) {
    if(low < high) {
        int loc = partition(nums, low, high);
        QuickSort(nums, low, loc - 1);
        QuickSort(nums, loc + 1, high);
    }
}
int main() {
    int n;
    cin >> n;//输入数组的长度
    vector<int> a(n);//定义数组
    for (int i = 0; i < n; i++) {//对数组初始化
        a[i]= -1 * (n + 1) + rand()%(2 * n + 2); //在 [-n-1,n+1] 随机产生数组
        cout << a[i] << " ";  //打印该数组
    }
    printf("\n");
    clock_t startTime = clock();//计时开始
    QuickSort(a, 0, n - 1); //手撕归并排序, 20000000 的数据量耗时 8.165s
    //sort(a.begin(), a.end()); //调 STL 库, 20000000 的数据量耗时 3.775s
    clock_t endTime = clock();//计时结束
    for(int i = 0; i < n; i++) {
        cout << a[i] << " "; //输出排序后的数组
    }
    printf("\n");
    cout << " 快速排序算法的运行时间为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

Problem 2

用长分别为 10000、30000、50000、80000、100000、200000 的 6 个数组 (可用机器随机产生) 的排列来统计这两种算法的时间复杂性;

Solution:

在上述两段代码中, 分别输入 $n = 10000, 30000, 50000, 80000, 100000, 200000$, 可得到归并排序算法的运行时间为 2ms, 6ms, 12ms, 19ms, 23ms, 49ms. 而对应手撕的快速排序算法的运行时间为 1ms, 4ms, 6ms, 10ms, 12ms, 26ms. STL 库中 sort 算法的运行时间为 1ms, 5ms, 6ms, 11ms, 14ms, 29ms. 由此可见, 当数据量比较大时, 快速排序算法会比归并排序更快! 值得一提的是, 当数据量为 20000000 时, 手撕快排、归并排序以及 STL 的 sort 算法的用时分别为 8.325s, 5.953s, 3.743s, 由此可见 STL 标准库的算法优化是很成熟的.

Problem 3

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 4

证明算法 **PartSelect** 的平均时间复杂性为 $O(n)$.

证明. 假定数组中的元素各不相同, 且第一次划分时划分元素 v 是第 i 小元素的概率为 $1/n$. 由于一趟快排的时间复杂度为 $O(n)$(因为是双指针解法), 所以不妨设一趟快排用时为 cn. 设 $C_A^k(n)$ 表示在数组 A 的 n 个元素中寻找第 k 小元素的平均时间复杂度. 若 $i = 1, \cdots, k-1$(即 $i < k$), 则子问题的平均用时分别为 $C_A^{k-i}(n-i)$; 若 $i = k+1, \cdots, n$(即 $i > k$), 则子问题的平均用时分别为 $C_A^k(i-1)$[1]. 于是, $C_A^k(n)$ 作为随机变量 v 的函数, 其数学期望表达式为 (即将概率与随机变量的函数取值相乘求和)

$$\begin{aligned} C_A^k(n) &\le cn + \frac{1}{n}\cdot C_A^{k-1}(n-1) + \cdots + \frac{1}{n}\cdot C_A^1(n-k+1) + \frac{O(1)}{n} + \frac{1}{n}\cdot C_A^k(k) + \cdots + \frac{1}{n}\cdot C_A^k(n-1) \\ &= cn + \frac{1}{n}\sum_{i=1}^{k-1} C_A^{k-i}(n-i) + \frac{1}{n}\sum_{i=k+1}^{n} C_A^k(i-1) \end{aligned}$$

令 $R(n) = \max\limits_k \left\{C_A^k(n)\right\}$, 下面通过数学归纳法来证明 $R(n) \le 4cn$:

- 当 $n = 1$ 时, 保证 $C \le 4c$, 即可满足 $R(1) = \max\limits_k \left\{C_A^k(1)\right\} = C \le 4c$;

[1] 当 $i = k$ 时, 显然子问题的平均用时就是 $O(1)$(即一下就找到了). 并且随机变量的函数取值就是子问题的平均用时.

- 设 $\forall m < n, R(m) \leq 4cm$ 都成立, 且不妨设 $R(n) = \max\limits_{k}\left\{C_A^k(n)\right\} = C_A^{k_n}(n)$, 数学期望不等式两边同时关于 k 取 max 即有如下[2]:

$$R(n) \leq cn + \frac{1}{n}\cdot\{R(n-1)+\cdots+R(n-k_n+1)\} + \frac{1}{n}\cdot\{R(k_n)+\cdots+R(n-1)\} \tag{1}$$

$$\leq cn + \frac{4c}{n}\cdot\{(n-1)+\cdots+(n-k_n+1)\} + \frac{4c}{n}\cdot\{k_n+\cdots(n-1)\} \tag{2}$$

$$\leq cn + \frac{4c}{n}\cdot\left\{\frac{(2n-k_n)(k_n-1)}{2} + \frac{(n+k_n-1)(n-k_n)}{2}\right\} \tag{3}$$

$$= cn + \frac{4c}{n}\cdot\left\{-k_n^2 + (1+n)k_n + \frac{n^2-3n}{2}\right\} \leq cn + \frac{4c}{n}\cdot\{\cdots\}_{k_n=\frac{n+1}{2}} \tag{4}$$

$$= cn + \frac{4c}{n}\cdot\left\{\frac{3n^2-4n+1}{4}\right\} \leq cn + c\cdot\{3n-3\} \leq 4cn \tag{5}$$

综上所述, 根据数学归纳法可得知 $\forall n \geq 1$, 都有 $R(n) \leq 4cn$, 即 **PartSelect** 算法的平均时间复杂度为 $O(n)$. □

Problem 5

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};
```

其中 **upper_bound** 是 STL 标准库里的函数, 其作用是在一段有序数组中寻找第一个严格大于 target 的元素位置[3]. 其具体算法的伪代码也如下所示:

[2]注意 (4) 式中第二项是关于 k_n 的二次函数 (开口向下), 所以可以放缩到他的最大值.
[3]是个迭代器, 若想得到下标, 则需要减去 A.begin()

Algorithm 2 二分查找 **upperbound** 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 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          ▷ 若是 lowerbound 算法, 此处判断条件则是 A[mid] < target
        low = mid + 1;
    else
        high = mid − 1;
    end if
end while
return low;                           ▷ 返回所要求的下标
end {upperbound}
```

所用到的 **upperbound** 函数和 **BinarySort** 函数编码如下:

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int upperbound(vector<int>& nums, int low, int high, int target) {
    while(low <= high) {
        int mid = (low + high) >> 1;
        if(nums[mid] <= target) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}
void BinarySort(vector<int>& nums) {
    int len = nums.size();
    for(int i = 1; i < len; i++) {
        int temp = nums[i];
        int low = upperbound(nums, 0, i - 1, temp); //自己手撕的二分查找 upperbound
        //int low = upper_bound(nums.begin(), nums.begin() + i, temp) - nums.begin(); //直接
↪   调用 STL 标准库的二分查找 upper_bound
        if(low != i) {
            for(int j = i - 1; j >= low; j--) {
                nums[j + 1] = nums[j];
            }
            nums[low] = temp;
        }
    }
}
```

紧接着主函数的编码如下 (经过调试, 读者可亲自运行):

```cpp
int main()
{
    int n;
    scanf("%d", &n);
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        a[i]= -1 * (n + 1) + rand()%(2 * n + 2);
        cout << a[i] << " ";
    }
    printf("\n");
    printf("\n");
    clock_t startTime = clock();
    BinarySort(a);
    clock_t endTime = clock();
    for(int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    printf("\n");
    printf("\n");
    cout << " 运行时间为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

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 6

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 (时空复杂度均为 $O(n)$): 最简单的做法是, 直接对数组做一次遍历, 将负数放入数组 temp1、将正数放入 temp2, 最后将 temp1 数组和 temp2 进行合并即可. 其算法伪码如下[4]: 可以看出, 该算法需要借助长度综合为 n 的临时数组, 所以空间复杂度为 $O(n)$, 算法只对数组 A 做了一次遍历, 因此时间复杂度也为 $O(n)$. 并且该算法不仅可以将负数放在左边、把正数放在右边, 还可以把 0 放在中间 (虽然功能多余了). 因此, 该问题可以拓展为: 设 A 是一个长度为 n 的数组, 要求设计时间复杂度为 $O(n)$ 的算法, 使得在数组 A 中**原地交换元素**[5]来使得数组 A 的负数排在左边、0 排在中间、正数排在右边. 该问题就是著名的荷兰三色国旗问题, 该问题的算法求解思想可以借助了 (三路) 快速排序 (即二路随机快排排序的进一步算法优化) 中的一趟快排操作.[6]

[4]由于该伪代码所对应的 C++ 程序基本一样, 所以此处就不给出对应的 C++ 代码了.

[5]即不允许算法借助其他临时数组, 即要求算法的空间复杂度为 $O(1)$.

[6]双路随机快速排序的缺点是: 当在数组中碰到大量的重复数元素时, 双路快排的许多次交换操作就显得很多余了.

Algorithm 3 双色旗问题的 **doublecolor** 算法

Input: n 个非 0 实数构成的数组 $A[0,\cdots,n-1]$
Output: 负数排在正数前面的数组 $A[0,\cdots,n-1]$

```
vector<int> temp1, temp2, temp3;
for int i = 0; i < A.size(); i++ do
    if A[i] < 0 then
        temp1.push_back(A[i]);
    else if A[i] = 0 then
        temp2.push_back(A[i]);
    else
        temp3.push_back(A[i]);
    end if
end for
copy(temp1.begin(), temp1.end(), A.begin());
copy(temp2.begin(), temp2.end(), A.begin() + temp1.size());
copy(temp3.begin(), temp3.end(), A.begin() + temp1.size() + temp2.size());
return A;
end {doublecolor}
```

方法 2 (时空复杂度分别为 $O(n), O(1)$): 我们先给出三路快速排序的算法伪码和 C++ 代码.

Algorithm 4 三路快速排序 **QuickSort3** 算法

Input: 数组 $A[0,\cdots,n-1]$ 的子段 $A[\text{left}, \text{right}]$
Output: 排序后的数组子段 $A[\text{left}, \text{right}]$

```
if left >= right then
    return ;
end if
int rand_index = left + rand()%(right - left + 1);
swap(A[left], A[rand_index]);
int pivot = A[left];
int lt = left − 1, i = left, gt = right + 1;
while i < gt do
    if A[i] == pivot then
        i++;
    else if A[i] > pivot then
        swap(A[i], A[gt − 1]), gt- -;
    else if A[i] < pivot then
        swap(A[lt + 1], A[i]), lt++, i++;
    end if
end while
QuickSort3(A, left, lt);
QuickSort3(A, gt, right);
end {QuickSort3}
```

三路快速排序的算法的 C++ 代码描述如下, 在主函数里输入 `QuickSort3(a, 0, n − 1)` 即可调用以排序数组. 一趟三路快速排序后的数组情况: 左边全是小于 pivot, 中间等于 pivot, 右边大于 pivot. 因此针对

荷兰三色国旗问题, 我们可以直接设置 pivot 为 0, 并将算法4的 1,2,3,4,5,17,18 行删去, 即可写出三色问题的求解算法, 具体见算法5.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
void QuickSort3(vector<int>& nums, int left, int right) {
    if(left >= right) {
        return;
    }
    int rand_index = left + rand()%(right - left + 1);
    swap(nums[left], nums[rand_index]);
    int pivot = nums[left];
    int lt = left - 1, i = left, gt = right + 1;
    while(i < gt) {
        if(nums[i] == pivot) {
            i++;
        }
        else if(nums[i] > pivot) {
            swap(nums[i], nums[gt - 1]);
            gt--;
        }
        else if(nums[i] < pivot) {
            swap(nums[lt + 1], nums[i]);
            lt++, i++;
        }
    }
    QuickSort3(nums, left, lt);
    QuickSort3(nums, gt, right);
}
```

Algorithm 5 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($A[i]$, $A[\text{gt}-1]$), gt- -;
  else if $A[i]$ < pivot **then**
    swap($A[\text{lt}+1]$, $A[i]$), lt++, i++;
  end if
end while
end {ThreeColor}

在此页的 C++ 程序上做对应的修改即可得到该算法的 C++ 程序, 由于作业空间限制, 此处就不予展示了.

Problem 7

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.

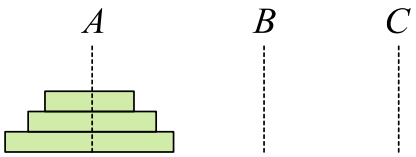

图 1: 汉诺塔问题

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6:

Algorithm 6 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

```
1: if n == 1 then
2:     move (A, C);
3: else
4:     Hanoi(A, B, n − 1);
5:     move (A, C);
6:     Hanoi(B, C, n − 1);
7: end if
8: 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

$$T(n) = 2T(n-1) + 1 \tag{6}$$

于是通过递推得到

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

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 8

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[7]:

Algorithm 7 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

```
1:  if n == 3 then                          ▷ 因为已知 L 为单峰数组, 因此 n ≥ 3
2:      return L[1];                        ▷ 若 n = 3, 则显然 L[1] 为峰顶
3:  else
4:      int low = 0, high = n − 1;
5:      while low <= high do
6:          int mid = (low + high) >> 1;
7:          if L[mid − 1] < L[mid] > L[mid + 1] then
8:              return L[mid];
9:          else if L[mid − 1] < L[mid] < L[mid + 1] then
10:             low = mid + 1;
11:         else
12:             high = mid − 1;
13:         end if
14:     end while
15: end if
16: 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 $T(n) = O(\log n)$.

[7] 具体的 C++ 程序代码见下一页

寻找单峰数组峰顶的 C++ 程序 (已在对应的**LeetCode 题目**上全部通过 42 个测试样例):

```cpp
class Solution {
public:
    int peakIndexInMountainArray(vector<int>& arr) {
        int n = arr.size(), res = 0;
        if(n == 3) {
            res = 1;
        }
        else {
            int low = 1, high = n - 2;  //起始指针不要放两边, 否则会越界
            while(low <= high) {
                int mid = (low + high) >> 1;
                if(arr[mid - 1] < arr[mid]) {
                    low = mid + 1;
                }
                else if(arr[mid - 1] > arr[mid]) {
                    high = mid - 1;
                }
            }
            res = high;  //此时 low > high, L[high] 就是对应的峰顶
        }
        return res;
    }
};
```

Problem 9

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:

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 8 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

```
1:  int low = 0, high = n − 1;
2:  while low <= high do
3:      int mid = (low + high) >> 1;
4:      if A[mid] < target then
5:          low = mid + 1;
6:      else
7:          high = mid − 1;
8:      end if
9:  end while
10: return low;                                  ▷ 返回所要求的下标
11: 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 10

在 n 枚 ($n \geq 3$) 硬币中有一枚重量不合格的硬币 (过轻或过重), 如果只有一架天平可以用来称重且称重的硬币数没有限制. 设计一个算法, 找出这枚不合格的硬币, 使得称重的次数最少 (优于 $O(n)$).

(提示: 分成 $n/3$ 或 $n/4$ 份, 至少两份数量相等)

Solution:

算法思想描述: 采用分治策略, 如果剩下的硬币个数小于 3(即 $n < 3$), 则将其逐个与正常硬币 (可以从拿走的硬币中随便选一个) 相比较, 不等的那枚就是不合格的硬币; 若 $n \geq 3$(即剩下的硬币个数至少是 3), 则将这些硬币分成 3 份, 其中两份个数一样 ($k = \lfloor n/3 \rfloor$), 另一份个数为 $n - 2k$. 将前两份称重对比, 若天平失衡则这两份中包含着异常硬币, 否则异常硬币在剩下的一份中. 递归进行划分处理, 直到 $n < 3$ 为止. 具体的算法伪代码见如下:

Algorithm 9 硬币检验算法 **CoinDet**(A, n)

Input: 共 n 枚硬币 (且含有 1 枚异常硬币) 的集合 A
Output: 1 枚异常硬币

```
1:  k := ⌊n/3⌋;
2:  if n < 3 then
3:      将其逐个与正常硬币 (从拿走的硬币中选一个) 相比较并找到异常硬币;        ▷ 递归终止条件
4:  else
5:      将集合 A 分成 3 份 X, Y, Z, 其中 |X| = |Y| = k, |Z| = n − 2k;
6:      if W(X) ≠ W(Y) then                                  ▷ W(X), W(Y) 分别为 X, Y 的重量
7:          A := X ∪ Y;
8:      else
9:          A := Z;
10:     end if
11:     n := |A|;
12:     CoinDet(A, n);
13: end if
14: end {CoinDet}
```

根据上述伪代码我们可以写出最坏情形下的时间复杂度 $T(n)$ 的递归方程[8]:

$$T(n) = T(2n/3) + O(1) \tag{7}$$

显然 $a = 1, b = 3/2, d = 0$, 故根据主定理可知 $T(n) = O(\log n)$.

最好情形下的时间复杂度 $T(n)$ 的递归方程为

$$T(n) = T(n/3) + O(1) \tag{8}$$

显然最好情形下 $a = 1, b = 3, d = 0$, 根据主定理可求得对应的时间复杂度也是 $O(\log n)$. 故不论最好最坏情形, 算法的时间复杂度都是 $O(\log n)$, 显然优于 $O(n)$.

[8] 子问题的规模要么是 Z 的规模 $n/3$(最好情形), 要么是 $X \cup Y$ 的规模 $2n/3$(最坏情形)

Problem 11

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 1 (基于哈希集合): 算法思想描述: 先用一个哈希集合存储数组 A 的元素[9], 然后遍历数组 B, 如果 B 中的元素在 A 对应的哈希集合中可以找到, 那么将该元素放进哈希集合 res 中, 最后将 res 转换为 vector 数组即可. 算法对应的 C++ 代码如下 (已在对应的**LeetCode T349**上全部通过 55 个测试样例):

```cpp
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> res;  //res 为哈希集合是为了对结果进行去重
        unordered_set<int> nums(nums1.begin(), nums1.end());
        for(auto num : nums2) {
            if(nums.find(num) != nums.end()) {  //如果找到了, 就放到 res 里
                res.insert(num);
            }
        }
        return vector<int>(res.begin(), res.end());  //将去重后得到的结果转为 vector 数组
    }
};
```

现在来分析一下此方法的时空复杂度: 使用一个集合存储数组 A 中的元素需要 $O(n)$ 的时间, 遍历集合 B 并判断元素是否在 A 的哈希集合中需要 $O(m)$ 的时间, 因此总的时间复杂度为 $O(m+n) = O(n)$. A 的哈希集合所占用的空间为 $O(n)$, 故空间复杂度为 $O(n)$.

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 10 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 $O(\log m) = O(\log \log n)$.

[9]哈希集合中的元素查找的时间复杂度是 $O(1)$, 且天然起到一个去重的作用

Problem 12

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 11 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: $\max\limits_{|S_1|=|S_2|}\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$

```
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 13

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 $r=3$, 则 3 个元素一组的中间值 u 是该数组的第 2 小元素, 此数组至少有 2 个小于等于 u; $\lfloor n/3 \rfloor$ 个中间值中至少有 $\lceil \lfloor n/3 \rfloor /2 \rceil$ 个小于等于这些中间值的中间值 v. 因此, 数组 A 中至少有 $2*\lceil \lfloor n/3 \rfloor /2 \rceil \geq \lfloor n/3 \rfloor \geq n/3-1$ 个元素小于等于 v. 即 A 中至多有 $n-(n/3-1)=2n/3+1$ 个元素大于 v. 同理, 至多有 $2n/3+1$ 个元素小于 v. 这样, 以 v 为划分元素所产生的新数组中至多有 $2n/3+1$ 个元素. 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

$$T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+cn \tag{9}$$

画出 $T(n)$ 的递归树, 见如下图2:

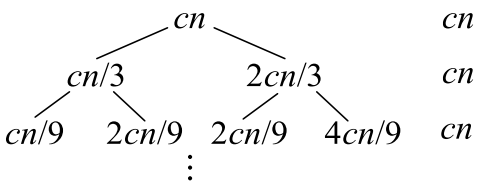

图 2: 递归树-1

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 $r=7$, 则 7 个元素一组的中间值 u 是该数组的第 4 小元素, 此数组至少有 4 个小于等于 u; $\lfloor n/7 \rfloor$ 个中间值中至少有 $\lceil \lfloor n/7 \rfloor /2 \rceil$ 个小于等于这些中间值的中间值 v. 因此, 数组 A 中至少有 $4*\lceil \lfloor n/7 \rfloor /2 \rceil \geq 2*\lfloor n/7 \rfloor \geq 2(n/7-1)$ 个元素小于等于 v. 即 A 中至多有 $n-2(n/7-1)=5n/7+2$ 个元素大于 v. 同理, 至多有 $5n/7+2$ 个元素小于 v. 这样, 以 v 为划分元素所产生的新数组中至多有 $5n/7+2$ 个元素. 既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

$$T(n)=T\left(\frac{n}{7}\right)+T\left(\frac{5n}{7}\right)+cn \tag{10}$$

画出 $T(n)$ 的递归树, 见如下图3:

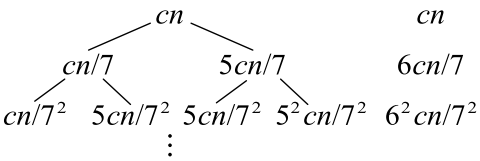

图 3: 递归树-2

故总的时间复杂度为:

$$T(n)=cn\left(1+\frac{6}{7}+\left(\frac{6}{7}\right)^2+\cdots\right)\leq cn\cdot\frac{1}{1-\frac{6}{7}}=7cn=O(n) \tag{11}$$

Problem 14

在 Internet 上的搜索引擎经常需要对信息进行比较, 比如可以通过某个人对一些事物的排名来估计他对各种不同信息的兴趣. 对于不同的排名结果可以用逆序来评价他们之间的差异. 考虑 $1, 2, \cdots, n$ 的排列 $i_1, i_2, ..., i_n$, 如果其中存在 i_j, i_k, 使得 $j < k$ 但 $i_j > i_k$, 那么就称 (i_j, i_k) 是这个排列的一个逆序. 一个排列含有逆序的个数称为这个排列的逆序数.

例如: 排列 $(2, 6, 3, 4, 5, 1)$ 含有 8 个逆序: (2,1),(6,3),(6,4),(6,5),(6,1),(3,1),(4,1),(5,1), 它的逆序数就是 8. 一个由 $1, 2, \cdots, n$ 组成的所有 $n!$ 个排列中, 最小的逆序数是 0, 对应的排列是 $(1, 2, 3, 4, \cdots, n)$, 最大的逆序数是 $n(n-1)/2$, 对应的排列是 $(n, n-1, \cdots, 2, 1)$. 逆序数越大的排列与原始排列的差异度越大. 利用二分归并排序算法设计一个计数给定排列逆序数的分治算法, 并对算法的时间复杂度进行分析.

Solution:

算法思想描述: 在二分归并排序算法中**顺带做了**计数逆序的工作, 在递归调用算法分别对子数组 L_1, L_2 排序时, 分别计算每个子数组内部的逆序; 在归并排好序的子数组 L_1, L_2 的过程中, 计算 L_1 的元素与 L_2 的元素之间产生的逆序. 具体来说, 合并两个有序数组所采用的是双指针算法, 每当遇到当前左指针 i 所指元素 $L_1[i] >$ 当前右指针 j 所指元素 $L_2[j]$ 时, 意味着 L_1 中从指针 i 到末尾的所有元素都与 $L_2[j]$ 构成逆序对, 于是此情形的逆序对增量为 $\text{mid} - i + 1$; 而当 $L_1[i] <= L_2[j]$ 时, 则此情形对于逆序对增量的贡献为 0(即没有逆序对). 算法伪代码见如下:

Algorithm 12 双指针合并算法 **Merge**(low, mid, high)

Input: 排序后的子数组 $A[\text{low}, \cdots, \text{mid}]$ 和 $A[\text{mid}+1, \cdots, \text{high}]$ ▷ 双指针法合并两个有序数组
Output: 合并好的升序数组 $A[\text{low}, \cdots, \text{high}]$ 及逆序数

```
1:  global count = 0;                                          ▷ count 是 int 型全局变量
2:  int k = 0, i = low, j = mid + 1;          ▷ i, j 是拣取游标 (即双指针), k 是向 B 存放元素的游标
3:  vector<int> B(high − low + 1);                             ▷ 借用临时数组 B
4:  while i ≤ mid && j ≤ high do                               ▷ 当两个集合都没有取尽时
5:      if A[i] ≤ A[j] then
6:          B[k++] := A[i++];                                  ▷ 此情形对逆序数的增量是没有贡献的
7:      else
8:          B[k++] := A[j++];
9:          count += mid − i + 1;    ▷ 此情形下, L1 中从指针 i 到末尾的所有元素都与 L2[j] 构成逆序对
10:     end if
11: end while
12: while i ≤ mid do                                  ▷ 当第二子组元素被取尽, 而第一组元素未被取尽时
13:     B[k++] := A[i++];
14: end while
15: while j ≤ high do                                 ▷ 当第一子组元素被取尽, 而第二组元素未被取尽时
16:     B[k++] := A[j++];
17: end while
18: copy(B.begin(), B.end(), A.begin() + low)          ▷ 将临时数组 B 中的元素拷贝给数组 A[low, ··· , high]
19: end{Merge}
```

Algorithm 13 归并排序主程序伪码 **MergeSort**(low, high)

Input: 待排序的数组 A 及下标 low,high
Output: 排序后的数组 $A[\text{low}, \cdots, \text{high}]$ 及 A 的逆序数

```
1: if low < high then
2:     int mid := ⌊(low + high)/2⌋          ▷ 求当前数组的分割点
3:     MergeSort(low, mid);                  ▷ 递归处理前半部分子数组
4:     MergeSort(mid + 1, high);             ▷ 递归处理后半部分子数组
5:     Merge(low, mid, high);                ▷ 归并两个排序后的子数组并计算逆序数
6: end if
7: end{MergeSort}
```

该算法对应的 C++ 程序 (已在对应的**LeetCode 题目**上全部通过 139 个测试样例) 为:

```cpp
class Solution {
public:
    int count; //定义为全局变量
    int reversePairs(vector<int>& nums) {
        count = 0;
        MergeSort(nums, 0, nums.size() - 1);
        return count;
    }

    void Merge(vector<int>& nums, int low , int mid, int high) {
        int i = low, j = mid + 1, k = 0;
        vector<int> temp(high - low + 1);
        while(i <= mid && j <= high) {
            if(nums[i] <= nums[j])
                temp[k++] = nums[i++];  //此情形下逆序数没有增量
            else {
                temp[k++] = nums[j++];
                count += mid - i + 1;  //此情形下逆序数才能有增量
            }
        }
        while(i <= mid) temp[k++] = nums[i++];  //把剩下的 nums 左半数组接上
        while(j <= high) temp[k++] = nums[j++];  //把剩下的 nums 右半数组接上
        copy(temp.begin(), temp.end(), nums.begin() + low);  //把临时数组的元素拷贝回 nums
        vector<int>().swap(temp);  //清除容器并最小化它的容量
    }

    void MergeSort(vector<int>& nums, int low, int high) {
        if(low < high) {
            int mid = (low + high) >> 1;
            MergeSort(nums, low, mid);
            MergeSort(nums, mid + 1, high);
            Merge(nums, low, mid, high);
        }
    }
};
```

显然该逆序数计算的算法伪码和 C++ 代码跟归并排序算法几乎一模一样, 就比归并排序在第 9 行多了个 count 的加法运算, 但算法的关键操作还是比较. 所以此算法的时间复杂度递推公式跟归并排序

一样:

$$T(n) = 2T(n/2) + O(n) \tag{12}$$

易知 $a = 2, b = 2, d = 1$, 故根据主定理可知 $T(n) = O(n \log n)$.

Problem 15

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[10]. 那么第 j 组 $(j = 1, 2, \cdots, \sqrt{n})$ 所含有的高度有

$$(j-1)\sqrt{n}+1, (j-1)\sqrt{n}+2, \cdots, (j-1)\sqrt{n}+\sqrt{n}, \quad j = 1, 2, \cdots, \sqrt{n} \tag{13}$$

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

$$T(n) = O(\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n}) \tag{14}$$

[10] 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 16

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

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

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:

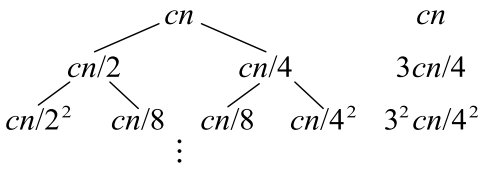

图 4: 递归树

故总的时间复杂度为:

$$T(n) = cn\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \cdots\right) \leq cn \cdot \frac{1}{1-\frac{3}{4}} = 4cn = O(n) \tag{15}$$

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

$$T(n) = T(n-1) + \log 3^n = T(n-2) + \log 3^{n-1} + \log 3^n \tag{16}$$

$$\cdots = T(1) + \log 3^2 + \log 3^3 + \cdots + \log 3^n \tag{17}$$

$$= 1 + \log\left(3^{2+3+\cdots+n}\right) = 1 + \log\left(3^{(n+2)(n-1)/2}\right) = \Theta\left(n^2\right) \tag{18}$$

(2). 易知

$$T(n) = T(n-1) + \frac{1}{n} = T(n-2) + \frac{1}{n-1} + \frac{1}{n} \tag{19}$$

$$\cdots = T(1) + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n} = \sum_{i=1}^{n}\frac{1}{i} = \Theta(\gamma + \log n) = \Theta(\log n) \tag{20}$$

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