算法设计与分析第一次作业

杨博文 21373037

2023 年 10 月 1 日

1. 计算渐进上界

1.

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

根据递推式推导可得： $T(n) = T(n-2) + 1 = T(n-4) + 2... = n/2$

因此 $T(n) = O(n)$

2.

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

根据主定理可得： $a = 1, b = 2, k = 1, k > \log_b a$

因此 $T(n) = O(n)$

3.

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

根据主定理可得： $a = 1, b = 3, k = 2, k > \log_b a$

因此 $T(n) = O(n^2)$

4.

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

根据递推式可得： $T(n) = T(n-1) + n^2 = T(n-2) + 2n^2 = T(n-3) + 3n^2... = T(n-n+1) + (n-1)n^2 = n^3 - n^2 + 1$

因此 $T(n) = O(n^3)$

5.

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

根据递推式可得： $T(n) = T(n-1) + 2^n = T(n-2) + 22^n ... = T(1) + (n-1)2^n = n2^n - 2^n + 1$

因此 $T(n) = O(n2^n)$

6.

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

根据主定理可得： $a = 1, b = 2, \log_b a = 0$**，然而** $\forall \epsilon > 0, \log n$ **的渐进小于** n^{ϵ}**，渐进大于** $n^{-\epsilon}$**，不能使用主定理。而** $\log n = \theta(\log n)$**，使用扩展形式主定理可解决。**

因此 $T(n) = O(\log^2 n)$

7.

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

根据主定理可得： $a = 4, b = 3, k = 1, k < \log_b a$

因此 $T(n) = O(n^{\log_3 4})$

2．k 路归并问题

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。

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）。

合并前两个数组需要 O(n+n) 时间，合并后的数组和第三个合并需要 O(2n+n) 时间...

总共需要 $2n + 3n + ... + kn = \frac{(2+k)(k-1)}{2}n = O(nk^2)$

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。

Algorithm 1 K 路归并排序

Input: $a_1[n], a_2[n]...a_k[n]$，将其顺序放入一个数组 $re[nk]$

Output: $re[nk]$

```
// 归并
Merge(left, mid, right):
leftMargin ← left * n, midMargin ← mid * (n + 1), rightMargin ← right * (n + 1)
tmp[leftMargin..rightMargin − 1] ← re[leftMargin..rightMargin − 1]
i ← leftMargin, j ← midMargin, k1 ← 0
while i < midMargin and j < rightMargin do
    if tmp[i] > tmp[j] then
        re[leftMargin + k1] ← tmp[j]
        k1 ← k1 + 1, j ← j + 1
    else
        re[leftMargin + k1] ← tmp[i]
        k1 ← k1 + 1, i ← i + 1
    end
end
while i < midMargin do
    re[leftMargin + k1] ← tmp[i]
    k1 ← k1 + 1, i ← i + 1
end
while j < rightMargin do
    re[leftMargin + k1] ← tmp[j]
    k1 ← k1 + 1, j ← j + 1
end
// 拆分
MergeSort(left, right):
 if left >= right then
    return
end
mid ← ⌊(left+right)/2⌋
MergeSort(left, mid)
MergeSort(mid + 1, right)
Merge(left, mid, right)
// 主函数入口
Main: MergeSort(0, k − 1)
```

3. Analysis

由于输入数组的状态即为局部有序数组，所以可利用二路归并的思想，看作是第 1 至 K 个有序数组进行二路归并，此时**分**所用时即为递归树深度 $\log k$，而合并不再是数组下标之间的局部数组进行合并，而是所对应的数个数组进行合并，每层递归树合并所用时间为 kn，因此该算法时间复杂度为 $O(kn\log k)$。

3. 三余因子和问题

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。

1. Algorithm

Algorithm 2 三余因子和问题

Input: 区间左端点 $left$, 区间右端点 $right$

Output: $result$

// 核心算法

Sum(left, right):

$re \leftarrow 0$

if $l > r$ **then**
  return *0*
end

if $l \equiv r$ *and* $(l \equiv 1$ *or* $l \equiv 2)$ **then**
  return l
end

for $i \leftarrow l; i <= r; i \leftarrow i + 1$ **do**
  if $i \bmod 3 \neq 0$ **then**
    $re \leftarrow re + i$
  end
end

$lnew \leftarrow l \bmod 3 \neq 0\ ?\ \lfloor \frac{l}{3} \rfloor + 1 : \frac{l}{3}$, $rnew \leftarrow \lfloor \frac{r}{3} \rfloor$

return $re + Sum(lnew, rnew)$

Main(): $result \leftarrow Sum(left, right)$

2. Analysis

本题求区间 [left, right] 上最大无法被 3 整除的因子之和，显然当数为非 3 的倍数时，其自身即为所求，而为 3 的倍数时，需要一直除 3 直到余数不是 3 的倍数，此刻即为所求。本算法将待求区间边界不断除 3 来缩小范围，特别注意临界返回条件，其递推式为 $T(n) = T(n/3) + O(n)$，根据主定理 $a = 1$，$b = 3$，$k = 1 > log_b a$，可计算出时间复杂度为 $O(n) \quad (n = right - left)$

4. 填数字问题

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 $A[[\frac{l+r}{2}]]$ 赋值为 i，其中 x 表示对数 x 做向下取整。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。

1. Algorithm

Algorithm 3 填数字问题

Input: $arrays\ a[n+1]$

Output: $a[1:n]$

// 将全为 0 的数组 a 按规则填满数

$Max \leftarrow 0,\ RightMargin \leftarrow 0,\ LeftMargin \leftarrow 0$

MergeSort(left, right):

if $left > right\ or\ (left \equiv right\ and\ a[left] \neq 0)$ **then**
  return
end

$mid \leftarrow \lfloor \frac{left+right}{2} \rfloor$

if $a[mid] \equiv 0$ **then**
  if $max < right - left + 1$ **then**
    $max \leftarrow right - left + 1$
    $LeftMargin \leftarrow left$
    $RightMargin \leftarrow right$
  end
else
  $MergeSort(left, mid-1)$
  $MergeSort(mid+1, right)$
end

// 主函数入口

$Main:$

for $i = 1;\ i <= n;\ i \leftarrow i+1$ **do**
  $LeftMargin \leftarrow 0,\ RightMargin \leftarrow 0,\ max \leftarrow 0$
  $MergeSort(1, n)$
  $a[\lfloor \frac{RightMargin+LeftMargin}{2} \rfloor] \leftarrow i$
end

打印 $a[1:n+1]$ 即为所求

2. Analysis

本题粗略推理后不难发现最常规的解法，即按照流程从 1 至 n 第一次遍历，在每次遍历中再遍历整个数组找到长度最长且连续为 0 的区间，时间复杂度为 $O(n^2)$。

进一步思考，从 1 到 n 的第一次整体遍历是必要的，因为从 1 至 n 都要分配进数组，因此尝试将寻找长度最长且连续为 0 的区间的时间复杂度优化为 $O(\log n)$。考虑到每次都是二分寻找 (即 $\frac{(left+right)}{2}$，与二路归并的思路类

似，这给了我们优化空间。我们只分不治，在分的时候若 $a[mid]=0$，说明此时 $[left, right]$ 内皆为 0，我们与全局最大长度比较并返回，最后得到长度最长且连续 0 的区间 leftMargin 和 rightMargin。每次寻找递归深度为 $\log n$，合并时所需时间复杂度为 $O(1)$，递归表达式为 $T(n) = 2T(\frac{n}{2}) + O(1)$，根据主定理，寻找区间操作时间复杂度为 $O(n)$，总体时间复杂度仍为 $O(n^2)$，下文进一步优化。

3. Algorithm

Algorithm 4 填数字问题$_2$

Input: n

Output: $a[1..n]$

// *LIST* 内存三元组或结构体，记录区间的左端点，右端点，长度

$LIST[n] \leftarrow \{\ \}$, $a[1..n] \leftarrow \{0\}$

SplitArray(left, right):

if $left > right$ **then**

  return ϕ

end

$mid \leftarrow \lfloor \frac{left+right}{2} \rfloor$

$LIST.append((left, right, right - left + 1))$

$SplitArray(left, mid - 1)$

$SplitArray(mid + 1, right)$

Main():

$SplitArray(1, n)$

将 LIST 按照先长度从大到小，若长度一致则左端点从小到大的顺序排序!!!

for $i \leftarrow 0$; $i < len(LIST)$; $i \leftarrow i + 1$ **do**

  $left \leftarrow LIST[i].left$, $right \leftarrow LIST[i].right$

  $a[\lfloor \frac{left+right}{2} \rfloor] \leftarrow i + 1$

end

打印 $a[1..n]$ 即为所求

4. Analysis

该算法先将所有的可能区间划分出来，时间复杂度根据前算法可知为 $O(n)$；接着为无序结构体数组排序，可使用快速排序，归并排序等方法，时间复杂度为 $O(n \log n)$，排好的顺序即按照规则填数的顺序；最后遍历排过序的有序结构体数组，在相应位置处填数，时间复杂度为 $O(n)$，由于这三步没有嵌套，总复杂度为 $O(n \log n)$。

5. 数字消失问题

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 $n = 2^k - 1$。

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。

1. Algorithm

Algorithm 5 数字消失问题

Input: n, $a[1..n]$

Output: $loss$

FindLoss():

```
flag[n+1] ← {0}
for i ← 0; i < n; i ← i+1 do
    flag[a[i]] ← 1
end
for i ← 0; i <= n; i ← i+1 do
    if flag[i] ≡ 0 then
        return i
    end
end
```

2. Analysis

使用哈希表第一次遍历记录出现过的数字，第二次遍历找到消失的数字，每次遍历时间复杂度为 $O(n)$，总时间复杂度为 $O(n)$。

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。

1. Algorithm

Algorithm 6 数字消失问题$_2$

Input: $k,\ A[1..2^k-1]$

Output: $loss$

FindLoss():

$zeroArr[n] \leftarrow \{0\},\ oneArr[n] \leftarrow \{0\}$

$zeroNum \leftarrow 0,\ oneNum \leftarrow 0,\ arrNum \leftarrow 2^k-1$

for $i \leftarrow 1;\ i <= k;\ i \leftarrow i+1$ **do**

  for $j \leftarrow 1;\ j <= arrNum;\ j \leftarrow j+1$ **do**

    if $bit-lookup[j][i] \equiv 0$ **then**

      $zeroNum \leftarrow zeroNum + 1$

      $zeroArr[zeroNum] \leftarrow A[j]$

    else

      $oneNum \leftarrow oneNum + 1$

      $oneArr[oneNum] \leftarrow A[j]$

    end

  end

  if $zeroNum > oneNum$ **then**

    $arrNum \leftarrow oneNum$

    $A[1..oneNum] \leftarrow oneArr[1..oneNum]$

  else

    $arrNum \leftarrow zeroNum$

    $A[1...zeroNum] \leftarrow zeroArr[1..zeroNum]$

  end

  $loss[i] \leftarrow zeroNum > oneNum\ ?\ 1 : 0$

  $zeroNum \leftarrow 0,\ oneNum \leftarrow 0$

  return $loss[1..k]$

end

2. Analysis

本问题主要思路为遍历数组中的每一二进制位，统计数组中每一二进制位 0 和 1 的多少，由于缺失一个数，二者个数一定差 1，设较小的一方为 x，目前为第 y 位，则缺失值的第 y 位即为 x，并将第 y 位为 x 的数组作为遍历第 y+1 位的子数组，直至 k 遍结束。每二进制位遍历中嵌套遍历子数组，则时间复杂度为 $2^k(TraverseArrayA) + 2^{k-1}(CopytheLessSubarraytoA) + 2^{k-1} + 2^{k-2}... = n + \frac{n}{2} + \frac{n}{2} + \frac{n}{4} + ... = 3n = O(n)$。