计算机算法设计与分析

083500M01001H

Chap 4 课程作业解答

2022 年 10 月 7 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

设有 n 个顾客同时等待一项服务. 顾客 i 需要的服务时间为 $t_i, 1 \le i \le n$. 应该如何安排 n 个顾客的服务次序才能使总的等待时间达到最小? 总的等待时间是各顾客等待服务的时间的总和. 试给出你的做法的理由 (证明).

Solution: 我们使用贪心算法求解该问题, 具体的**贪心策略**为: **服务时间较短的优先安排**. 假设调度 f 的顺序为 $i_1, i_2, \cdots, i_n$, 那么 i_k 的等待时间为 $\sum_{j=1}^{k-1} t_{i_j}$, 总的等待时间为

$$T(f) = \sum_{i=1}^{n} (n-i)\, t_{i_j} \tag{1}$$

根据贪心策略, 需要先排序使得 $t_1 \le t_2 \le \cdots \le t_n$, 按照 $1, 2, \cdots, n$ 的顺序安排服务. 则调度 f^* 的总等待时间为

$$T(f^*) = \sum_{i=1}^{n} (n-i)\, t_i \tag{2}$$

于是我们可以给出对应的算法伪码:

Algorithm 1 Service 算法

Input: 服务时间的数组 $T[1, \cdots, n] = [t_1, t_2, \cdots, t_n]$
Output: 调度 f, $f(i)$ 为第 i 个顾客的开始服务时刻, $1 \le i \le n$

sort(T.begin(), T.end()); ▷ 按照服务时间从小到大的顺序排列
$f(1) := 0$;
for $i := 2$ to n **do**
  $f(i) := f(i-1) + t_{i-1}$;
end for
return f;
end {Service}

由于**算法主要在于排序**, 故其最坏情况下的时间复杂度为 $O(n \log n)$.

下面证明: **对任何输入, 对服务时间短的顾客优先安排将得到最优解.**

证明. 使用**交换论证**的方法: 假设存在某个最优解 g 且存在 $i, j \in A, i < j$, 但是 i 在 j 之后得到服务. 那么在 g 的安排中一定存在相邻安排的顾客 i 和 j, 使得 $i < j$ 但 i 在 j 之后得到服务. 交换 i 和 j 得到新的服务顺序 g', 那么

$$T(g) - T\left(g'\right) = t_j - t_i \ge 0 \tag{3}$$

总的等待时间将不会增加, 因此 g' 也是最优解. 至多经过 $n(n-1)/2$ 次这样的交换, 就可以将 g 转换为算法的解 f^*, 从而证明了 f^* 是最优解. □

Problem 2

字符 $a \sim h$ 出现的频率分布恰好是前 8 个 Fibonacci 数, 它们的 Huffman 编码是什么? 将结果推广到 n 个字符的频率分布恰好是前 n 个 Fibonacci 数的情形. Fibonacci 数的定义为: $F_0 = 1, F_1 = 1, F_n = F_{n-2} + F_{n-1} (n \geq 1)$.

Solution: 对应的 Huffman 树如下图1所示. 故可知 Huffman 编码为

$$h : 0,\ g : 10,\ f : 110,\ e : 1110,\ d : 11110,\ c : 111110,\ b : 1111110,\ a : 1111111 \tag{4}$$

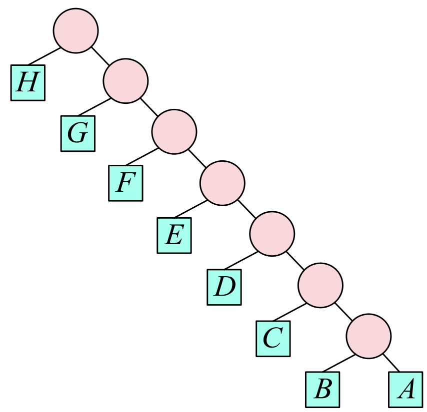

图 1: Huffman 树

为了推广, 需要先证明一个结论: 设 $f_i (i \geq 1)$ 为 Fibonacci 数列, 则

$$\sum_{i=1}^{k} f_i \leq f_{k+2} \tag{5}$$

证明. 采用数学归纳法: 当 $k = 1$ 时, 命题显然成立; 假设 $k = n$ 时命题成立, 则 $k = n + 1$ 时, 则有

$$\sum_{i=1}^{n+1} f_i = \sum_{i=1}^{n} f_i + f_{n+1} \leq f_{n+2} + f_{n+1} = f_{n+3} \tag{6}$$

于是 $\forall k \in \mathbb{N}^*$, 不等式 (5) 都成立. $\square$

因此根据上述结论, 前 k 个字符合并后子树的根权值小于等于第 $k+2$ 个 Fibonacci 数. 根据 Huffman 算法, 他将继续参加与第 $k+1$ 个字符的合并. 因此 n 个字符的 Huffman 编码按照频数从小到大依次为

$$\underbrace{11\cdots1}_{n-1\text{个}1},\ \underbrace{11\cdots10}_{n-2\text{个}1},\ \underbrace{11\cdots0}_{n-3\text{个}1},\ \cdots,\ 10,\ 0 \tag{7}$$

即第 $i (i > 1)$ 个字母的编码为 $\underbrace{11\cdots10}_{n-i\text{个}1}$.

Problem 3

设 $p_1, p_2, \cdots, p_n$ 是准备存放到长为 L 的磁带上的 n 个程序, 程序 p_i 需要的带长为 a_i. 设 $\sum_{i=1}^{n} a_i > L$, 要求选取一个能放在带上的程序的最大子集合 (即其中含有最多个数的程序)Q. 构造 Q 的一种贪心策略是按 a_i 的非降次序将程序计入集合.

(1). 证明这一策略总能找到最大子集 Q, 使得 $\sum_{p_i \in Q} a_i \le L$;

(2). 设 Q 是使用上述贪心算法得到的子集合, 磁带的利用率可以小到何种程度?

(3). 试说明 (1) 中提到的设计策略不一定能得到使 $\sum_{p_i \in Q} a_i / L$(即磁带的利用率) 取最大值的子集合.

Solution:

(1).

证明. 还是采用**交换论证**: 易知只要存放程序名称相同 (不管次序) 的任何方法都是同样的解. 不妨设最优解为 $\mathrm{OPT} = \{i_1, i_2, \cdots, i_j\}, i_1 < i_2 < \cdots < i_j, j < n$. 如果

$$\{i_1, i_2, \cdots, i_j\} = \{1, 2, \cdots, j\} \tag{8}$$

那么算法的解就是最优解. 假设 $\{i_1, i_2, \cdots, i_j\} \ne \{1, 2, \cdots, j\}$, 设 $i_1 = 1, i_2 = 2, \cdots, i_{t-1} = t-1, i_t > t$. 用 t 替换 i_t, 那么得到的解 I^* 占用的存储空间与解 OPT 占用空间的差值为

$$S(I^*) - S(\mathrm{OPT}) = a_t - a_{i_t} \le 0 \tag{9}$$

因此 I^* 也是最优解, 但是它比解 OPT 减少了一个标号不相等的程序. 对于解 OPT, 从第一个标号不等的程序开始, 至多经过 j 次替换, 就得到最优解 $\{1, 2, \cdots, j\}$. 显然算法的时间复杂度为

$$T(n) = O(n \log n) + O(n) = O(n \log n) \tag{10}$$

□

(2). 磁带的利用率最小可以小到 0, 比如 $\forall 1 \le i \le n, a_i > L$.

(3). 按照题中的贪心策略虽然能够保障所装的程序最多, 但对应的空间利用率不一定最大. 具体例子为: 设 $\{a_1, a_2, \cdots a_s\}$ 为 Q 的最大子集, 可能会有: 用 a_{s+1} 替换 a_s, 子集合变为 $\{a_1, a_2, \cdots a_{s-1}, a_{s+1}\}$ 并且满足 $\sum_{k=1}^{s-1} a_k + a_{s+1} < L$. 虽然程序个数仍为 s 个, 但利用率却增加了. 因此上述贪心策略并不一定能求得使利用率最大化的最优解.

(4). 如果要求磁带利用率最大, 这个问题的本质上时 0-1 背包问题, 每个程序相当于物品, 其重量和价值就是所需要的存储带长, 背包的重量限制等于磁带容量 L. 可以使用动态规划 (DP) 算法来解决: 设 $F_k(y)$ 表示考虑前 k 个程序, 磁带空间为 y 时的最大存储量. 递推方程为

$$F_k(y) = \begin{cases} \max\{F_{k-1}(y), F_{k-1}(y - a_k) + a_k\}, a_k \le y \le L \\ F_{k-1}(y), a_k > y \end{cases} \tag{11}$$

其中 $k > 0$ 且 $F_0(y) = 0\,(0 \le y \le L), F_k(0) = 0, F_k(y) = -\infty\,(y < 0)$. 可以设定如下标记函数 $i_k(y)$ 用于追踪解:

$$i_k(y) = \begin{cases} k, \text{若} F_{k-1}(y) \le F_{k-1}(y - a_k) + a_k, a_k \le y \le L\,(k > 1) \\ i_{k-1}(y), \text{否则} \end{cases}, \quad i_1(y) = \begin{cases} 1, \text{若} y \ge a_1 \\ 0, \text{否则} \end{cases} \tag{12}$$

该 DP 算法在最坏情形下的时间复杂度为 $T(n) = O(nL)$.

Problem 4

写出 Huffman 编码的伪代码, 并编程实现.

Solution: 伪代码见如下算法2:

Algorithm 2 HuffmanCode 算法

Input: 待编码的数组 $A[1, \cdots, n]$
Output: 数组 A 的 Huffman 编码

local h; ▷ 最小化堆, 内含元素为结点类型, 堆初始为空
int i;
Node p, q, r; ▷ 结点数据结构, 内含数值以及分别指向左、右儿子的两个指针
for $i = 1; i <= n; i{+}{+}$ **do** ▷ 将数组 A 中的所有元素插入堆
  Insert$(h, A[i])$;
end for
while $|h| > 1$ **do** ▷ h 元素个数大于 1
  $p =$ **DeleteMin**(h); $q =$ **DeleteMin**(h); ▷ 移除最小的两个结点
  $r = p + q$; r.left $= \min(p, q)$; r.right $= \max(p, q)$; ▷ 构造新的结点 r, 其值为 p, q 值之和
  Insert(h, r); ▷ 将 r 插入堆 h 中
end while
$p =$ **DeleteMin**(h); ▷ 取出最后一个结点, 此节点即为 Huffman 树的根节点
end {HuffmanCode}

其对应的 C++ 程序代码见如下:

```cpp
#include<iostream>
#include<string>
using namespace std;

struct Node
{
    double weight;
    string ch;
    string code;
    int lchild, rchild, parent;
};

void Select(Node huffTree[], int *a, int *b, int n)//找权值最小的两个 a 和 b
{
    int i;
    double weight = 0; //找最小的数
    for (i = 0; i < n; i++)
    {
        if (huffTree[i].parent != -1)     //判断节点是否已经选过
            continue;
        else
        {
            if (weight == 0)
            {
                weight = huffTree[i].weight;
```

```
                *a = i;
            }
            else
            {
                if (huffTree[i].weight < weight)
                {
                    weight = huffTree[i].weight;
                    *a = i;
                }
            }
        }
    }
    weight = 0; //找第二小的数
    for (i = 0; i < n; i++)
    {
        if (huffTree[i].parent != -1 || (i == *a))//排除已选过的数
            continue;
        else
        {
            if (weight == 0)
            {
                weight = huffTree[i].weight;
                *b = i;
            }
            else
            {
                if (huffTree[i].weight < weight)
                {
                    weight = huffTree[i].weight;
                    *b = i;
                }
            }
        }
    }
    int temp;
    if (huffTree[*a].lchild < huffTree[*b].lchild)  //小的数放左边
    {
        temp = *a;
        *a = *b;
        *b = temp;
    }
}

void Huff_Tree(Node huffTree[], int w[], string ch[], int n)
{
    for (int i = 0; i < 2 * n - 1; i++) //初始过程
    {
        huffTree[i].parent = -1;
        huffTree[i].lchild = -1;
        huffTree[i].rchild = -1;
        huffTree[i].code = "";
    }
```

```
    for (int i = 0; i < n; i++)
    {
        huffTree[i].weight = w[i];
        huffTree[i].ch = ch[i];
    }
    for (int k = n; k < 2 * n - 1; k++)
    {
        int i1 = 0;
        int i2 = 0;
        Select(huffTree, &i1, &i2, k); //将 i1, i2 节点合成节点 k
        huffTree[i1].parent = k;
        huffTree[i2].parent = k;
        huffTree[k].weight = huffTree[i1].weight + huffTree[i2].weight;
        huffTree[k].lchild = i1;
        huffTree[k].rchild = i2;
    }
}

void Huff_Code(Node huffTree[], int n)
{
    int i, j, k;
    string s = "";
    for (i = 0; i < n; i++)
    {
        s = "";
        j = i;
        while (huffTree[j].parent != -1) //从叶子往上找到根节点
        {
            k = huffTree[j].parent;
            if (j == huffTree[k].lchild) //如果是根的左孩子，则记为 0
            {
                s = s + "0";
            }
            else
            {
                s = s + "1";
            }
            j = huffTree[j].parent;
        }
        cout << " 字符 " << huffTree[i].ch << " 的编码: ";
        for (int l = s.size() - 1; l >= 0; l--)
        {
            cout << s[l];
            huffTree[i].code += s[l]; //保存编码
        }
        cout << endl;
    }
}
```

```
string Huff_Decode(Node huffTree[], int n,string s)
{
    cout << " 解码后为: ";
    string temp = "", str = "";//保存解码后的字符串
    for (int i = 0; i < s.size(); i++)
    {
        temp = temp + s[i];
        for (int j = 0; j < n; j++)
        {
            if (temp == huffTree[j].code)
            {
                str = str + huffTree[j].ch;
                temp = "";
                break;
            }
            else if (i == s.size()-1 && j == n-1 && temp != "")//全部遍历后没有
            {
                str = " 解码错误! ";
            }
        }
    }
    return str;
}

int main()
{
    //编码过程
    const int n = 5;
    Node huffTree[2 * n];
    string str[] = { "A", "B", "C", "D", "E"};
    int w[] = { 30, 30, 5, 20, 15 };
    Huff_Tree(huffTree, w, str, n);
    Huff_Code(huffTree, n);
    //解码过程
    string s;
    cout << " 输入编码: ";
    cin >> s;
    cout << Huff_Decode(huffTree, n, s) << endl;
    system("pause");
    return 0;
}
```

Problem 5

设有一条边远山区的道路 AB, 沿着道路 AB 分布着 n 所房子. 这些房子到 A 的距离分别是 $d_1, d_2, \cdots,$ $d_n (d_1 < d_2 < \cdots < d_n)$. 为了给所有房子的用户提供移动电话服务, 需要在这条道路上设置一些基站. 为了保证通讯质量, 每所房子应该位于距离某个基站的 4km 范围内. 设计一个算法找基站的位置, 并且使得基站的总数最少, 并证明算法的正确性.

Solution: 使用贪心法, 令 $a_1, a_2, \cdots$ 表示基站的位置. 贪心策略为: 首先令 $a_1 = d_1 + 4$. 对 $d_2, d_3, \cdots, d_n$ 依次检查, 找到下一个不能被该基站覆盖的房子. 如果 $d_k \le a_1 + 4$ 但 $d_{k+1} > a_1 + 4$, 那么第 $k+1$ 个房子不能被基站覆盖, 于是取 $a_2 = d_{k+1} + 4$ 作为下一个基站的位置. 照此下去, 直到检查完 d_n 为止. 伪代码见如下算法3:

Algorithm 3 Location 算法

Input: 距离数组 $d[1, \cdots, n] = [d_1, d_2, \cdots, d_n]$, 满足 $d[1] < d[2] < \cdots < d[n]$
Output: 基站位置的数组 a

```
a[1] := d[1] + 4; k := 1;
for j = 2; j <= n; j++ do
    if d[j] > a[k] + 4 then
        a[++k] := d[j] + 4;
    end if
end for
return a;
end {Location}
```

结论: 对任何正整数 k, 存在最优解包含算法前 k 步选的的基站位置.

证明. $k = 1$, 存在最优解包含 $a[1]$. 如若不然, 有最优解 OPT, 其第一个位置是 $b[1]$ 且 $b[1] \ne a[1]$, 那么 $d_1 - 4 \le b[1] < d_1 + 4 = a[1]$. $b[1]$ 覆盖的是距离在 $[d_1, b[1] + 4]$ 之间的房子. $a[1]$ 覆盖的是距离在 $[d_1, a[1] + 4]$ 的房子. 因为 $b[1] < a[1]$, 且 $b[1]$ 覆盖的房子都在 $a[1]$ 覆盖的区域内, 故用 $a[1]$ 替换 $b[1]$ 得到的仍是最优解;

假设对于 k, 存在最优解 A 包含算法前 k 步选择的基站位置, 即

$$A = \{a[1], a[2], \cdots, a[k]\} \cup B \tag{13}$$

其中 $a[1], a[2], \cdots, a[k]$ 覆盖了距离为 $d_1, d_2, \cdots, d_j$ 的房子. 那么 B 是关于 $L = \{d_{j+1}, d_{j+2}, \cdots, d_n\}$ 的最优解. 否则, 存在关于 L 的更优解 B^*, 那么用 B^* 替换 B 就会得到 A^* 且 $|A^*| < |A|$, 这与 A 是最优解相矛盾. 根据归纳假设可得知 L 有一个最优解 $B' = \{a[k+1], \cdots\}, |B'| = |B|$. 于是

$$A' = \{a[1], a[2], \cdots, a[k]\} \cup B' = \{a[1], a[2], \cdots, a[k], a[k+1], \cdots\} \tag{14}$$

且 $|A'| = |A|$, 故 A' 也是最优解, 从而命题对于 $k + 1$ 也成立. 故根据数学归纳法可知, 对任何正整数 k 命题都成立. □

算法的关键操作是 for 循环, 而循环体内部的操作都是常数时间, 因此算法在最坏情况下的时间复杂度为 $O(n)$.

Problem 6

有 n 个进程 $p_1, p_2, .., p_n$, 进程 p_i 的开始时间为 $s[i]$, 截止时间为 $d[i]$. 可以通过检测程序 Test 来测试正在运行的进程, Test 每次测试时间很短, 可以忽略不计, 即如果 Test 在时刻 t 测试, 那么它将对满足 $s[i] \le t \le d[i]$ 的所有进程同时取得测试数据. 问: 如何安排测试时刻, 使得对每个进程至少测试一次, Test 测试的次数达到最少? 设计算法并证明正确性, 分析算法复杂度.

Solution: 贪心策略: 将进程按照 ddl 进行排序. 取第 1 个进程的 ddl 作为第一个测试点, 然后顺序检查后续能够被这个测试点检测的进程 (这些进程的开始时间 $\le$ 测试点), 直到找到下一个不能被测试到的进程为止. 伪码见如下算法4:

Algorithm 4 Test 算法

Input: 开始时间的数组 $s[1, \cdots, n]$, 截止时间的数组 $d[1, \cdots, n]$
Output: 数组 t: 顺序选定的测试点构成的数组

将进程按照 $d[i]$ 递增的顺序进行排序 (使得 $d[1] \le d[2] \le \cdots \le d[n]$);
$i := 1; t[i] := d[1]; j := 2$ ▷ 第一个测试点是最早结束进程的 ddl
while $j \le n$ && $s[j] \le t[i]$ **do** ▷ 检查进程 j 是否可以在时刻 $t[i]$ 被测试
    j++;
end while
if $j > n$ **then**
    return t;
else
    $t[++i] := d[j++]$, **goto** 3; ▷ 找到待测进程中结束时间最早的进程 j
end if
end {Test}

结论: 对于任意正整数 k, 存在最优解包含算法前 k 步选择的测试点.

证明. $k = 1$ 时, 设 $S = \{t[i_1], t[i_2], \cdots\}$ 是最优解, 不妨设 $t[i_1] < t[1]$. 设 p_u 是在时刻 $t[i_1]$ 被测到的任意进程, 那么 $s(u) \le t[i_1] \le d[u]$, 从而有

$$s[u] \le t[i_1] < t[1] = d[1] \le d[u] \tag{15}$$

因此 p_u 也可以在 $t[1]$ 时刻被测试. 于是在 S 中用 $t[1]$ 替换掉 $t[i_1]$ 后也可得到一个最优解.

假设对于任意 k, 算法在前 k 步选择了 k 个测试点 $t[1], t[i_2], \cdots, t[i_k]$ 且存在最优解

$$T = \{t[1], t[i_2], \cdots, t[i_k]\} \cup T' \tag{16}$$

设算法前 k 步选择的测试点不能测到的进程构成集合 $Q \subseteq P$, 其中 P 为全体进程集合. 不难证明 T' 是子问题 Q 的最优解[1]. 根据归纳假设可得知, $\exists Q$ 的最优解 T^* 包含测试点 $t[i_{k+1}]$, 即

$$T^* = \{t[i_{k+1}]\} \cup T'' \tag{17}$$

因此有

$$\{t[1], t[i_2], \cdots, t[i_k]\} \cup T^* = \{t[1], t[i_2], \cdots, t[i_{k+1}]\} \cup T'' \tag{18}$$

也是原问题的最优解, 根据归纳法可知命题成立. □

算法的时间复杂度为 $T(n) = O(n \log n) + O(n) = O(n \log n)$.

[1]反证法: 假设 T' 不是子问题 Q 的最优解, 则会推出 T 不是最优解, 显然矛盾.

Problem 7

设有作业集合 $J = \{1, 2, \cdots, n\}$, 每项作业的加工时间都是 1, 所有作业的截止时间是 D. 若作业 i 在 D 之后完成, 则称为被延误的作业, 需赔偿罚款 $m(i)(i = 1, 2, \cdots, n)$, 这里 D 和 $m(i)$ 都是正整数, 且 n 项 $m(i)$ 彼此不等. 设计一个算法求出使总罚款最小的作业调度算法, 证明算法的正确性并分析时间复杂度.

Solution: 贪心策略: 优先安排前 D 个罚款最多的作业. 正确性证明需要利用交换论证的方法, 先给出以下结论:

结论: 设作业调度 f 的安排次序是 $\langle i_1, i_2, \cdots, i_n \rangle$, 那么罚款为

$$F(f) = \sum_{k=D+1}^{n} m(i_k) \tag{19}$$

证明. 显然最优调度没有空闲时间, 不妨假设作业是连续安排的. 因为每项作业的加工时间都是 1, 再截止时间 D 之前可以完成 D 项作业. 只有在 D 之后安排的 $n - D$ 项作业 (即 $i_{D+1}, i_{D+2}, \cdots, i_n$ 都是被罚款的作业). □

根据上述结论可以推出: 令 S 是 $n - D$ 项罚款最少的作业构成的集合.

(1). 对于 S(或 $J \backslash S$) 中的作业 i 和 j, 交换 i, j 的加工顺序不影响总罚款;

(2). 对于作业 i 和 $j, m(i) < m(j)$, 调度 f 将 i 安排在 D 之前, j 安排在 D 之后, 那么交换作业 i 和 j 得到的调度 g, 则 g 的罚款会减少, 这是因为

$$F(g) - F(f) = m(i) - m(j) < 0 \tag{20}$$

根据上述分析可以看出, 把罚款最小的 $n - D$ 项作业安排在最后会使得总罚款金额达到最小.

于是可以设计出以下算法5

Algorithm 5 Work 算法

Input: 罚款数组 $m[1, \cdots, n]$, 作业集合 J

Output: 作业调度 f

1: 利用 **PartSelect** 算法从 $m(1), m(2), \cdots, m(n)$ 中选出第 $n - D$ 小的元素 (记作 m^*);
2: 用 m^* 与数组 m 中剩下的 $n - 1$ 个元素进行比较, 找出比 m^* 小的 $n - D - 1$ 个元素;
3: 将步骤 2 中的元素和 m^* 所对应的作业从 D 时刻开始以任意顺序进行加工;
4: 将剩下的 D 项作业以任意顺序安排在 $0, 1, \cdots, D - 1$ 时刻加工;
5: **end {Work}**

现在来分析一下这个算法的时间复杂度: 第 1 行调用了 **PartSelect** 算法, 最坏需要 $O(n)$ 的时间[2]; 算法中的第 2 步对剩余数组元素做一次遍历也需要 $O(n)$ 的时间, 故总的时间复杂度为 $T(n) = O(n) + O(n) = O(n)$. 其实也可以先将作业按照 $m(i)$ 由大到小进行排序, 然后直接安排前 D 项作业即可, 但是排序所需的时间复杂度为 $O(n \log n)$, 效率上显然不如上述的 **work** 算法.

---

[2]**改进后**的 PartSelect 算法在最坏情形下的时间复杂度为 $O(n)$.

Problem 8

举出反例证明: 本章开始例 1 贪心规则找零钱算法 (目标: 零币数量最少; 规则: 尽量先找币值大的), 在零钱种类不合适时, 贪心算法结果不正确.

Solution: 比如, 如果提供找零的面值是 11,5,1, 找零 15. 使用贪心算法找零方式为 11+1+1+1+1, 需要五枚硬币. 而最优解为 5+5+5, 只需要 3 枚硬币.

至此, Chap 4 的作业解答完毕.