计算机算法设计与分析

083500M01001H

Chap 1-2 课程作业

2022 年 9 月 11 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

试确定下述程序的关键操作数, 该函数实现一个 $m \times n$ 矩阵与一个 $n \times p$ 矩阵之间的乘法:

```
1  template <class T>
2  void Mult(T **a, T **b, int m, int n, int p) {
3      //m×n 矩阵 A 与 n×p 矩阵 B 相成得到 m×p 矩阵 C
4      for(int i = 0; i < m; i++) {
5          for(int j = 0; j < p ;j++) {
6              T sum = 0;
7              Tfor(int k = 0; k < n; k++)
8              Tsum += a[i][k]*b[k][j];
9              C[i][j] = sum;
10         }
11     }
12 }
```

Solution:

此算法的关键操作为**第 8 行**, 该语句的操作数为 2 (1 次加法和 1 次乘法). 则三层循环的总关键操作数为:

$$T = \sum_{i=0}^{m-1} \sum_{j=0}^{p-1} \sum_{k=0}^{n-1} 2 = 2mnp$$

故该算法的时间复杂度为

$$T(m, n, p) = O(2mnp) = O(mnp)$$

Problem 2

函数 MinMax 用来查找数组 $a[0 : n-1]$ 中的最大元素和最小元素, 以下给出两个程序. 令 n 为实例特征. 试问: 在各个程序中, a 中元素之间的比较次数在最坏情况下各是多少?

```
/* 找最大最小元素 (方法一)*/
template <class T>
bool MinMax(T a[], int n, int& Min, int& Max) {
    //寻找 a[0:n-1] 中的最小元素与最大元素
    //如果数组中的元素数目小于 1, 则返回 false
    if(n<1) return false;
    Min=Max=0; //初始化
    for(int i=1; i<n; i++) {
        if(a[Min]>a[i]) Min=i;
        if(a[Max]<a[i]) Max=i;
    }
    return true;
}
```

```cpp
/* 找最大最小元素 (方法二)*/
template <class T>
bool MinMax(T a[], int n, int& Min, int& Max) {
    //寻找 a[0:n-1] 中的最小元素与最大元素
    //如果数组中的元素数目小于 1, 则返回 false
    if(n<1) return false;
    Min=Max=0; //初始化
    for(int i=1; i<n; i++) {
        if(a[Min]>a[i]) Min=i;
        else if(a[Max]<a[i]) Max=i;
    }
    return true;
}
```

Solution:

不论数组 a 是单调递增还是单调递减, for 循环内部的两次判断都得执行, 所以方法 1 在任何情况下的元素比较次数都为 $2 \times (n-1)$; 而对于方法 2 来说, 当数组 a 单调递减 (最好情况) 时, for 循环内部的第一个判断条件一定满足, 那么 else if 这个判断自然就不用执行了, 即此情形下的元素比较次数为 $1 \times (n-1)$. 但当数组 a 单调递增 (最坏情况) 时, 第一个循环条件在各轮循环中都不能满足, 所以紧跟着需要执行后边的 else if 判断, 即此情形下的元素比较次数为 $2 \times (n-1)$.

Problem 3

证明以下关系式不成立: (1). $10n^2+9=O(n)$; (2). $n^2\log n=\Theta\left(n^2\right)$.

Solution:

证明. **(1).** 考虑极限 $\lim\limits_{n\to\infty}\dfrac{10n^2+9}{n}=+\infty>c\ \ (\forall c>0)$, 故根据大 O 比率定理可知该式不成立;

(2). 考虑极限 $\lim\limits_{n\to\infty}\dfrac{n^2\log n}{n^2}=+\infty>c\ \ (\forall c>0)$, 故根据 Θ 比率定理可知该式不成立. □

Problem 4

按照渐近阶从低到高的顺序排列以下表达式:

$$4n^2, \log n, 3^n, 20n, n^{2/3}, n!$$

Solution:

根据 Stirling 公式:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \ (\text{as}\, n \to +\infty)$$

可知有渐进阶的顺序为

$$\log n \ll n^{2/3} \ll 20n = \Theta(n) \ll 4n^2 = \Theta\left(n^2\right) \ll 3^n \ll n!$$

Problem 5

(1). 假设某算法在输入规模是 n 时为 $T(n) = 3 * 2^n$. 在某台计算机上实现并完成该算法的时间是 t 秒. 现有另一台计算机, 其运行速度为第一台的 64 倍. 那么, 在这台计算机上用同一算法在 t 秒内能解决规模为多大的问题?

(2). 若上述算法改进后的新算法的时间复杂度为 $T(n) = n^2$, 则在新机器上用 t 秒时间能解决输入规模为多大的问题?

(3). 若进一步改进算法, 最新的算法的时间复杂度为 $T(n) = 8$, 其余条件不变, 在新机器上运行, 在 t 秒内能够解决输入规模为多大的问题?

Solution:

(1). 设问题规模为 M, 则新机器上的求解时间为 $t = 3 \times 2^M/64$, 老机器的求解时间 $t = 3 \times 2^n$, 即解得 $M = n + 6$;

(2). 同理设问题规模为 M, 则新机器上的求解时间为 $t = M^2/64 = n^2$, 老机器的求解时间 $t = n^2$, 即解得 $M = 8n$;

(3). 因为该算法的时间复杂度是常数阶的, 也就意味着问题规模不影响求解时间 (当问题规模很大时), 所以在任何 (可以运行该算法的) 机器上, t 秒内可以解决任意规模的问题.

Problem 6

考虑下述选择排序算法1所示:

Algorithm 1 选择排序

Input: n 个不等整数的数组 $A[1..n]$
Output: 按递增次序排序的 A

```
for i := 1 to n do
    for j := i + 1 to n do
        if A[j] < A[i] then
            A[i] ↔ A[j]
        end if
    end for
end for
输出排序后的数组 A
```

问: (1) 最坏情况下做多少次比较运算?
(2) 最坏情况下做多少次交换运算? 在什么输入时发生?

Solution:

(1). 任意情况下要比较 $(n-1)+(n-2)+\cdots+1=\frac{1}{2}n(n-1)$ 次. 再者, 此处不存在所谓的最好或最坏情况, 不论数组 A 是逆序还是升序排列, 计算机不可能因为提前得知 A 的所有情况而不执行判断语句, 即每次循环都需要进行比较运算;

(1). 最坏情况: 输入数组内元素为降序排列. 此时做 $(n-1)+(n-2)+\cdots+1=\frac{1}{2}n(n-1)$ 次交换运算. 当数组 A 内元素为升序排列时, 则交换次数为 0 (即为最好情况).

Problem 7

考虑下面的每对函数 $f(n)$ 和 $g(n)$, 比较他们的阶.

(1). $f(n)=\frac{1}{2}\left(n^2-n\right),\ g(n)=6n$; (2). $f(n)=n+2\sqrt{n},\ g(n)=n^2$;

(3). $f(n)=n+n\log n,\ g(n)=n\sqrt{n}$; (4). $f(n)=\log(n!),\ g(n)=n^{1.05}$;

Solution:

(1). $f(n)=\Theta\left(n^2\right)\gg\Theta(n)=g(n)$; **(2)**. $f(n)=\Theta(n)\ll\Theta\left(n^2\right)=g(n)$;

(3). $f(n)=\Theta(n\log n)\ll\Theta\left(n^{1.5}\right)=g(n)$; **(4)**. 根据**斯特林公式**可知:

$$\log(n!)\sim\log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)=\frac{1}{2}\log(2\pi n)+n(\log n-\log e)\sim n\log n\ \ (\text{as}\, n\to\infty)$$

所以有 $f(n)=\Theta(n\log n)\ll\Theta\left(n^{1.05}\right)=g(n)$.

Problem 8

在表1中填入 true 或 false.

	$f(n)$	$g(n)$	$f(n)=O(g(n))$	$f(n)=\Omega(g(n))$	$f(n)=\Theta(g(n))$
1	$2n^3+3n$	$100n^2+2n+100$			
2	$50n+\log n$	$10n+\log\log n$			
3	$50n\log n$	$10n\log\log n$			
4	$\log n$	$\log^2 n$			
5	$n!$	5^n			

表 1: 原始表格

Solution:

解答如下表2所示:

	$f(n)$	$g(n)$	$f(n)=O(g(n))$	$f(n)=\Omega(g(n))$	$f(n)=\Theta(g(n))$
1	$2n^3+3n$	$100n^2+2n+100$	False	True	False
2	$50n+\log n$	$10n+\log\log n$	True	True	True
3	$50n\log n$	$10n\log\log n$	False	True	False
4	$\log n$	$\log^2 n$	True	False	False
5	$n!$	5^n	False	True	False

表 2: 解答

Problem 9

用迭代法求解下列递推方程:

(1). $\begin{cases} T(n) = T(n-1) + n - 1 \\ T(1) = 0 \end{cases}$; (2). $\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + n - 1 \\ T(1) = 0 \end{cases}$, $n = 2^k$

Solution:

(1). 易知

$$\begin{cases} T(2) - T(1) = 1 \\ T(3) - T(2) = 2 \\ \vdots \\ T(n) - T(n-1) = n - 1 \end{cases} \xrightarrow{\text{各式相加}} T(n) = 1 + 2 + \cdots + n - 1 = \frac{1}{2}n(n-1) = \Theta\left(n^2\right)$$

(2). 令 $n = 2^k$, 则易知有如下:

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

至此, Chap 1-2 的作业解答完毕.

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

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[gt − 1]), gt--;
    else if A[i] < pivot then
        swap(A[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.

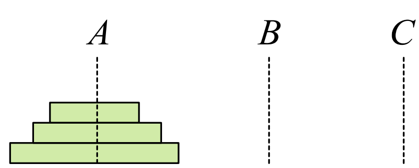

图 1: 汉诺塔问题

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 ≥ 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 [mid − 1] < L [mid] > L [mid + 1] then
            return L [mid];
        else if L [mid − 1] < L [mid] < L [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: $\max\limits_{|S_1|=|S_2|}\left|\sum_{x\in S_1} x - \sum_{x\in S_2} x\right|$

```
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 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:

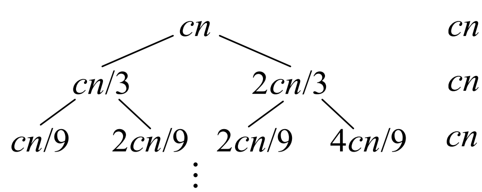

图 2: 递归树-1

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:

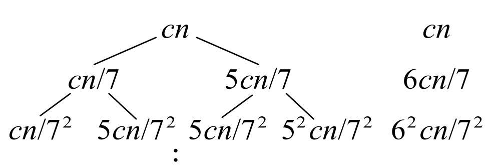

图 3: 递归树-2

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:

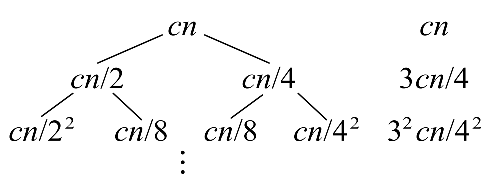

图 4: 递归树

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: $\gamma = \lim_{n\to+\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$.

University of Chinese Academy of Sciences

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 $T(f) = \sum_{i=1}^{n} (n-i)\, t_{i_j}$. 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 $T(f^*) = \sum_{i=1}^{n} (n-i)\, t_i$. 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. **交换论证:** 不妨设 $t_1 \le t_2 \le \cdots t_n$, 算法的调度 f 结果为 $1, 2, \cdots, n$. 如果它不是最优的, 则存在最优调度 f^*, 设其最早第 k 项作业 i_k 与 f 不同, 即 $f^* : 1, 2, \cdots, k-1, i_k, i_{k+1}, \cdots, i_n$. 则必有 $t_{i_k} \ge t_k$. 现将 f^* 中的作业 k 与作业 i_k 置换, 得到调度 $f^{**} : 1, 2, \cdots, k, i_{k+1}, \cdots, i_k, \cdots, i_n$. 其中 i_k 位置为 j, 则 $j > k, t_{i_k} \ge t_k$. 则有

$$T(f^*) - T(f^{**}) = (j-k)\,(t_{i_k} - t_k) \ge 0$$

说明 f^{**} 也是最优调度, 且它与 f 不同的次序项后移了一位. 重复最多 n 步, 则可得 f 最优. □

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

$$h : 0,\ g : 10,\ f : 110,\ e : 1110,\ d : 11110,\ c : 111110,\ b : 1111110,\ a : 1111111 \tag{1}$$

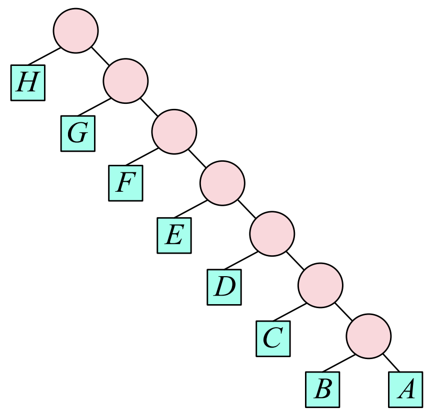

图 1: Huffman 树

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

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

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

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

于是 $\forall k \in \mathbb{N}^*$, 不等式 (2) 都成立. □

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

$$\underbrace{11\cdots1}_{n-1\text{个}1},\ \underbrace{11\cdots10}_{n-2\text{个}1},\ \underbrace{11\cdots0}_{n-3\text{个}1},\ \cdots,\ 10,\ 0 \tag{4}$$

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 $\sum\limits_{i=1}^{n} a_i > L$, 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 $\sum\limits_{p_i \in Q} a_i \leq L$;

(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 $\sum\limits_{p_i \in Q} a_i / L$(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 OPT $= \left\{i_1, i_2, \cdots, i_j\right\}, i_1 < i_2 < \cdots < i_j, j < n$. 如果

$$\left\{i_1, i_2, \cdots, i_j\right\} = \{1, 2, \cdots, j\} \tag{5}$$

那么算法的解就是最优解. 假设 $\left\{i_1, i_2, \cdots, i_j\right\} \neq \{1, 2, \cdots, j\}$, 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

$$S\left(I^*\right) - S\left(\mathrm{OPT}\right) = a_t - a_{i_t} \leq 0 \tag{6}$$

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

$$T\left(n\right) = O\left(n \log n\right) + O\left(n\right) = O\left(n \log n\right) \tag{7}$$

□

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

(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 $\sum\limits_{k=1}^{s-1} a_k + a_{s+1} < L$. 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, 这个问题的本质上是 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 $F_k\left(y\right)$ 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

$$F_k\left(y\right) = \begin{cases} \max\left\{F_{k-1}\left(y\right), F_{k-1}\left(y - a_k\right) + a_k\right\}, a_k \leq y \leq L \\ F_{k-1}\left(y\right), a_k > y \end{cases} \tag{8}$$

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

$$i_k\left(y\right) = \begin{cases} k, \text{若} F_{k-1}\left(y\right) \leq F_{k-1}\left(y - a_k\right) + a_k, a_k \leq y \leq L\left(k > 1\right) \\ i_{k-1}\left(y\right), \text{否则} \end{cases}, \quad i_1\left(y\right) = \begin{cases} 1, \text{若} y \geq a_1 \\ 0, \text{否则} \end{cases} \tag{9}$$

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}

Problem 5

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.

Problem 6

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 $b[1] \neq a[1]$, 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

$$A = \{a[1], a[2], \cdots, a[k]\} \cup B \tag{10}$$

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

$$A' = \{a[1], a[2], \cdots, a[k]\} \cup B' = \{a[1], a[2], \cdots, a[k], a[k+1], \cdots\} \tag{11}$$

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 7

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

```
1:  将进程按照 d[i] 递增的顺序进行排序 (使得 d[1] ≤ d[2] ≤ ··· ≤ d[n]);
2:  i := 1; t[i] := d[1]; j := 2                 ▷ 第一个测试点是最早结束进程的 ddl
3:  while j ≤ n && s[j] ≤ t[i] do               ▷ 检查进程 j 是否可以在时刻 t[i] 被测试
4:      j++;
5:  end while
6:  if j > n then
7:      return t;
8:  else
9:      t[++i] := d[j++], goto 3;               ▷ 找到待测进程中结束时间最早的进程 j
10: end if
11: 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

$$s[u] \le t[i_1] < t[1] = d[1] \le d[u] \tag{12}$$

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

$$T = \{t[1], t[i_2], \cdots, t[i_k]\} \cup T' \tag{13}$$

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

$$T^* = \{t[i_{k+1}]\} \cup T'' \tag{14}$$

因此有

$$\{t[1], t[i_2], \cdots, t[i_k]\} \cup T^* = \{t[1], t[i_2], \cdots, t[i_{k+1}]\} \cup T'' \tag{15}$$

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 8

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

$$F(f) = \sum_{k=D+1}^{n} m(i_k) \tag{16}$$

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

$$F(g) - F(f) = m(i) - m(j) < 0 \tag{17}$$

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: $m^* :=$ **Partselect**$(m[\ \ ], n - D)$;
2: **Partition**$(m[\ \ ], A, B, m^*)$;
3: $\{i_1, i_2, \cdots, i_D\} := B$;
4: $\{i_{D+1}, i_{D+2}, \cdots, i_n\} := A + \{m^*\}$;
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; 第 2 步调用的 **Partition** 算法也需要 $O(n)$ 的时间, 故总的时间复杂度为 $T(n) = O(n) + O(n) = O(n)$.

计算机算法设计与分析

083500M01001H

Chap 5 课程作业解答

2022 年 10 月 22 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

最大子段和问题: 给定整数序列 $a_1, a_2, \cdots, a_n$, 求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值:

$$\max\left\{0, \max_{1\le i\le i\le n} \sum_{k=i}^{j} a_k\right\}$$

(1). 已知一个简单算法如下:

```cpp
int Maxsum(int n, vector<int> a, int& besti, int& bestj) {
    int sum = 0;
    for(int i = 1; i <= n; i++) {
        int suma = 0;
        for(int j = i; j <= n; j++) {
            suma += a[j];
            if(suma > sum) {
                sum = suma;
                besti = i;
                bestj = j;
            }
        }
    }
    return sum;
}
```

试分析该算法的时间复杂性;

(2). 试用分治算法解最大子段和问题, 并分析算法的时间复杂性;

(3). 试说明最大子段和问题具有最优子结构性质, 并设计一个动态规划算法求解最大子段和问题, 分析算法的时间复杂度. (提示: 可令 $b(j) = \max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k, j = 1, 2, \cdots, n$)

Solution: (1). 显然第 2 层 for 循环里面的操作都是常数次的 (记为 C), 所以算法总的关键操作数为

$$\sum_{i=1}^{n}\sum_{j=i}^{n} C = C\sum_{i=1}^{n}(n-i+1) = \frac{1}{2}C\left(n^2+n\right)$$

故显然时间复杂度为 $T(n) = O(n^2)$.

(2). 采用分治算法, 则考虑: 先将数组从中间 mid 切开. 此时, 最大和的子段可能出现在左半边, 也可能出现在右半边, 也有可能横跨左右两个子数组. 所以需要返回这三种情况下所分别对应的子问题解的最大值.

当最大和的子段出现在左半边 (右半边同理) 时, 继续分中点递归直至分解到只有一个数为止;

当最大和的子段横跨 mid 左右时, 只需分别求解左子数组的最优后缀和以及右子数组的最优前缀和. 这三种情形下的最大值即为整个数组的最大子段和. 具体分治算法见如下.

Algorithm 1 MaxSubSum(A, left, right)

Input: 数组 A, 左边界 left, 右边界 right

Output: A 的最大子段和 sum 及子段的前后边界

```
if left==right then
    return max(A[left], 0);
end if
k :=(left+right)/2;
leftsum = MaxSubSum(A,left,k);
rightsum = MaxSubSum(A, k + 1,right);
类似 (1) 中的算法分别求得 S1, S2;
Sum:= S1 + S2;
return max(leftsum, rightsum, Sum);
end {MaxSubSum}
```

可以看出最坏情形下的时间复杂度的递推式和结果分别为 $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, 故根据主定理 $(\log_b(a) = \log_2(2) = 1 = d)$ 可知 $T(n) = O(n \log n)$.

(3). 先证明此问题具有最优子结构性质: 依次考虑 $(1 \le i \le n)$ 以 $a[i]$ **为结尾**的最大子段和 $C[i]$, 然后在这 n 个值当中取最大值即为原问题答案. 假设以 $a[i]$ 为结尾的最大 (和) 子段为 $\{a[k], \cdots, a[i]\}$, 那么 $\{a[k], \cdots, a[i-1]\}$ 一定是以 $a[i-1]$ 为结尾的最大 (和) 子段. 否则若 $\{a[m], \cdots, a[i-1]\}$ 为以 $a[i-1]$ 为结尾的最大 (和) 子段, 那么 $\{a[m], \cdots, a[i-1], a[i]\}$ 就是以 $a[i]$ 为结尾的最大 (和) 子段, 这显然与假设相矛盾, 也就是说该优化函数是满足优化原则的 (即此问题具有最优子结构性质).

现在来推导 $C[i]$ 的递推表达式: 当 $C[i-1] \le 0$, 说明 $C[i-1]$ 对应的子段对于整体的贡献是没有的, 所以 $C[i] \leftarrow a[i]$; 当 $C[i-1] > 0$, 说明 $C[i-1]$ 对应的子段对于整体的是有贡献的, 于是 $C[i] \leftarrow a[i] + C[i-1]$. 两种可能情况 (对应两种决策) 取最大值即可:

$$\begin{cases} C[i] = \max\{a[i], C[i-1] + a[i]\}, 2 \le i \le n \\ C[1] = a[1] \end{cases}$$

最后返回数组 C 中的最大值 ($\max\limits_{1 \le i \le n} C[i]$) 即可. 计算 $C[i]$ 的过程需要消耗 $O(n)$ 的时间, 找出数组最大值也需要 $O(n)$ 的时间 (一次遍历), 所以算法的总时间复杂度为 $T(n) = O(n)$. 并且我们可以给出 C++ 代码:

```cpp
int mostvalue(vector<int>& a) { //时间复杂度为 O(n), 空间复杂度为 O(n)
    int n = a.size();
    vector<int> dp(n);
    dp[0] = a[0];
    for(int i = 1; i < n; i++) {
        dp[i] = max(dp[i - 1] + a[i], a[i]);
    }
    int index = max_element(dp.begin(), dp.end()) - dp.begin();
    return dp[index];
}
```

Problem 2

设 $A=\{x_1,x_2,\cdots,x_n\}$ 是 n 个不等的整数构成的序列, A 的一个单调递增子序列是指序列 $\{x_{i_1},x_{i_2},\cdots,x_{i_k}\},i_1<i_2<\cdots i_k$ 且 $x_{i_1}<x_{i_2}<\cdots<x_{i_k}$.(子序列包含 k 个整数). 例如, $A=\{1,5,3,8,10,6,4,9\}$, 他的长度为 4 的递增子序列是: $\{1,5,8,10\},\{1,5,8,9\},\cdots$. 设计一个算法, 求 A 的最长的单调递增子序列, 分析算法的时间复杂度. 对于输入实例 $A=\{2,8,4,-4,5,9,11\}$, 给出算法的计算过程和最后的解.

Solution: 定义 $dp[i]$ 是以 nums$[i]$ **为结尾** (且考虑前 i 个元素) 的最长单增子序列的长度. 于是可以写出如下转移方程 ($i\geq 1$) 和初始条件:

$$dp\left[i\right]=\begin{cases}\max\limits_{0\leq j\leq i-1} dp\left[j\right]+1, & \exists j\in\left[0,i-1\right],s.t.\,\text{nums}\left[j\right]<\text{nums}\left[i\right]\\ 1 & \forall j\in\left[0,i-1\right],s.t.\,\text{nums}\left[j\right]>\text{nums}\left[i\right]\end{cases},dp\left[0\right]=1$$

显然, 若 $dp[i]$ 的子序列是 $i_1i_2\cdots i_ki$, 则 $dp[i_k]$ 的子序列为 $i_1i_2\cdots i_k$, 即问题满足最优子结构性质. 需借助数组 m 来对解进行回溯 (即 $m[i]$ 记录 $dp[i]$ 是由哪个下标的状态转移而来的). 而要想算出**整个数组**的最长单增子序列长度, 则需要算好所有的 $dp[i]$ 值, 再对 dp 数组进行遍历, 由此得到最长单增子序列长度和对应下标. 最后使用数组 m 进行回溯以取得答案. 可以看出, 算法的空间复杂度为 $O(n)$, 而时间复杂度显然为

$$T\left(n\right)=O\left(\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1\right)=O\left(\sum_{i=0}^{n-1}i\right)=O\left(\frac{n\left(n-1\right)}{2}\right)=O\left(n^2\right)$$

我们将上述的最优值求解过程和解的回溯过程写成 C++ 代码, 并且已完全通过**LeetCode-T300**的所有测试样例, 具体如下所示:

```cpp
vector<int> LIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 0);
    dp[0] = 1;
    vector<int> m(n, 0);
    for (int i = 1; i <= n - 1; i++) {
        int prev = i, len = 1;
        for (int j = 0; j <= i - 1; j++) {
            if (nums[j] < nums[i]) {
                if (dp[j] + 1 > len) {
                    len = dp[j] + 1;
                    prev = j;
                }
            }
        }
        dp[i] = len, m[i] = prev;
    }
    int index = max_element(dp.begin(), dp.end()) - dp.begin();
    int MaxLen = dp[index];
    vector<int> res;
    while (res.size() != MaxLen) {
        res.push_back(nums[index]);
        index = m[index];
    }
    return res;
}
```

对于具体实例, 计算过程如下:

$$C[1]=1; C[2]=2, k[2]=1; C[3]=2, k[3]=1; C[4]=1, k[4]=0;$$
$$C[5]=3, k[5]=3; C[6]=4, k[6]=5; C[7]=5, k[7]=6$$

显然在数组 C 中的最大值为 $C[7]=5$, 即最长递增子序列长度为 5 且追踪过程为:

$$x_7, k[7]=6 \Rightarrow x_6;\ k[6]=5 \Rightarrow x_5;\ k[5]=3 \Rightarrow x_3;\ k[3]=1 \Rightarrow x_1$$

故 $A=\{2,8,4,-4,5,9,11\}$ 的最长单调递增子序列为 $\{x_1,x_3,x_5,x_6,x_7\}=\{2,4,5,9,11\}$.

Problem 3

考虑下面特殊的整数线性规划问题

$$\max \sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_i x_i \le b, x_i \in \{0,1,2\}, 1 \le i \le n$$

试设计一个解决此问题的动态规划算法, 并分析算法的时间复杂度.

Solution: 这是整数背包问题, 下证最优子结构性质: 设 $y_1, y_2, \cdots y_n$ 是原问题的最优解, 则 $y_1, y_2, \cdots y_{n-1}$ 是下述子问题的最优解:

$$\max \sum_{i=1}^{n-1} c_i x_i, \text{s.t.} \sum_{i=1}^{n-1} a_i x_i \le b - a_n y_n, x_i \in \{0,1,2\}, 1 \le i \le n-1$$

如若不然, 设 $y_1', y_2', \cdots y_{n-1}'$ 是子问题的最优解, 则

$$\sum_{i=1}^{n-1} c_i y_i' > \sum_{i=1}^{n-1} c_i y_i \text{且} \sum_{i=1}^{n-1} a_i y_i' \le b - a_n y_n \Rightarrow \sum_{i=1}^{n-1} c_i y_i' + c_n y_n > \sum_{i=1}^{n} c_i y_i \text{且} \sum_{i=1}^{n-1} a_i y_i' + a_n y_n \le b$$

于是 $y_1', y_2', \cdots y_{n-1}', y_n$ 为原问题的最优解, 与 $y_1, y_2, \cdots y_n$ 是最优解相矛盾! 设 $m[k][x]$ 表示容量约束 x, 可装入 $1, 2, \cdots, k$ 件物品的最优值, 则有递推公式:

$$m[k][x] = \max\{m[k-1][x], m[k-1][x-a_k]+c_k, m[k-1][x-2a_k]+2c_k\}, 0 \le x \le b$$
$$m[0][x]=0, x \ge 0; \quad m[0][x]=-\infty, x<0; \quad m[k][<0]=-\infty, k=1,\cdots,n$$

Problem 4

可靠性设计: 一个系统由 n 级设备串联而成, 为了增强可靠性, 每级都可能并联了不止一台同样的设备. 假设第 i 级设备 D_i 用了 m_i 台, 该级设备的可靠性 $g_i(m_i)$, 则这个系统的可靠性是 $\Pi g_i(m_i)$. 一般来说 $g_i(m_i)$ 都是递增函数, 所以每级用的设备越多系统的可靠性越高. 但是设备都是有成本的, 假定设备 D_i 的成本是 c_i, 设计该系统允许的投资不超过 c. 那么, 该如何设计该系统 (即各级采用多少设备) 使得这个系统的可靠性最高. 试设计一个动态规划算法求解可靠性设计问题.

Solution: 问题描述为

$$\max \prod_{i=1}^{n} g_i\left(m_i\right), \text{s.t.} \sum_{i=1}^{n} m_i c_i \leq C$$

证明问题具有最优子结构性质: 设 $m_1, m_2, \cdots, m_{n-1}, m_n$ 是原问题的最优解, 其可靠性为 $\prod_{i=1}^{n} g_i\left(m_i\right) = g_n\left(m_n\right) \prod_{i=1}^{n-1} g_i\left(m_i\right)$, 则 $m_1, m_2, \cdots, m_{n-1}$ 显然是下述子问题的最优解:

$$\max \prod_{i=1}^{n-1} g_i\left(m_i\right), \text{s.t.} \sum_{i=1}^{n-1} m_i c_i \leq C - m_n c_n$$

否则若 $m_1', m_2', \cdots, m_{n-1}'$ 是子问题的最优解, 则 $\prod_{i=1}^{n-1} g_i\left(m_i'\right) > \prod_{i=1}^{n-1} g_i\left(m_i\right)$ 且 $\sum_{i=1}^{n-1} m_i' c_i \leq C - m_n c_n$. 于是有 $g_n\left(m_n\right) \cdot \prod_{i=1}^{n-1} g_i\left(m_i'\right) > \prod_{i=1}^{n} g_i\left(m_i\right)$ 且 $\sum_{i=1}^{n-1} m_i' c_i + m_n c_n \leq C$, 即 $m_1', m_2', \cdots, m_{n-1}', m_n$ 则为原问题的最优解, 这显然与 $m_1, m_2, \cdots, m_{n-1}, m_n$ 是原问题最优解相矛盾! 因而 $m_1, m_2, \cdots, m_{n-1}$ 是子问题的最优解. 即此问题具有最优子结构性质. 设 $m[k][x]$ 为成本 x, 前 k 级设备串联所得最优可靠性值, 则有:

$$\begin{aligned} & m[k][x] = \max\left\{m[k-1]\left[x - m_k c_k\right] \times g_k\left(m_k\right)\right\}, 1 \leq m_k \leq x / c_k \\ & m[0][x] = 1, x \geq 0;\ m[k]\left[c_k\right] = 1 \end{aligned}$$

```
Safe(c[], g[], C) {
    int m[][], p[][];
    C = c - sum(c);
    for(j = 0 to C) m[0][j] = 1;
    for(i = 1 to n) {
        for(j = 0 to c) {
            m[i][j] = 0;
            for(k = 0 to j/c[i]) {
                t = m[i-1][j-k*c[i]]*g[i](k);
                if(m[i][j] < t) {
                    m[i][j] = t, p[i][j] = k;
                }
            }
        }
    }
}
```

最优解是 $m[n][C]$, 可以如下回溯求 m_i:

$$p[n][C] = m_n, C = C - m_n \cdot c[n]; p[n-1][C] = m_{n-1}; \cdots$$

Problem 5

(**双机调度问题**) 用两台处理机 A 和 B 处理 n 个作业. 设第 i 个作业交给机器 A 处理时所需要的时间是 a_i, 若由机器 B 来处理, 则所需要的时间是 b_i. 现在要求每个作业只能由一台机器处理, 每台机器都不能同时处理两个作业. 设计一个动态规划算法, 使得这两台机器处理完这 n 个作业的时间最短 (从任何一台机器开工到最后一台机器停工的总时间). 以下面的例子说明你的算法:

$$n=6,(a_1,a_2,a_3,a_4,a_5,a_6)=(2,5,7,10,5,2),(b_1,b_2,b_3,b_4,b_5,b_6)=(3,8,4,11,3,4)$$

Solution: 在完成前 k 个作业时, 设机器 A 工作了 x 时间 (注意 x 限制为正整数), 则机器 B 此时最小的工作时间是 x 的函数. 设 $F[k][x]$ 表示完成前 k 个作业时, 机器 B 最小的工作时间, 则有

$$F[k][x]=\min\{F[k-1][x]+b_k,F[k-1][x-a_k]\}$$

其中 $F[k-1][x]+b_k$ 对应的是第 k 个作业由机器 B 来处理, 此时完成前 $k-1$ 个作业时机器 A 的工作时间仍是 x, 则 B 在 $k-1$ 阶段用时为 $F[k-1][x]$; 而 $F[k-1][x-a_k]$ 对应第 k 个作业由机器 A 处理 (完成 $k-1$ 个作业, 机器 A 工作时间时 $x-a[k]$, 而 B 完成 k 阶段与完成 $k-1$ 阶段用时都为 $F[k-1][x-a_k]$). 于是完成前 k 个作业所需要的时间为 $T=\max\{x,F[k][x]\}$. 根据上述递推关系很容易证得问题满足最优子结构性质. 并且通过下述算法代码可知时间复杂度为 $O\left(n\cdot\min\left\{\sum_{i=1}^{n}a_i,\sum_{i=1}^{n}b_i\right\}\right)$.

```
int schedule() {
    int sumA = a[1], time[n];
    //k = 1 的情况
    for(int x = 0; x < a[1]; x++) {
        F[1][x] = b[1];
    }
    F[1][a[1]] = min(b[1],a[1]);
    //初始化
    for(int i = 2; i <= n; i++) {
        for(int j = 0; j <= n; j++) {
            F[i][j] = INT_MAX;
        }
    }
    //k >= 2 的情况
    for(int k = 2; k <= n; k++) {
        sumA += a[k];
        time[k] = INT_MAX;
        for(int x = 0; x <= sumA; x++) {
            if(x < a[k]) {
                F[k][x] = F[k-1][x] + b[k];
            } else {
                F[k][x] = min(F[k-1][x] + b[k], F[k-1][x-a[k]]);
            }
            //判断完成作业 k 时, 到底是机器 B 所需最小时间小, 还是 A 所需时间小
            time[k] = min(time[k],max(x,F[k][x]));
        }
    }
    return time[n];
}
```

Problem 6

有 n 项作业的集合 $J = \{1, 2, \cdots, n\}$, 每项作业 i 有加工时间 $t(i) \in \mathbb{Z}^+, t(1) \le t(2) \le \cdots \le t(n)$, 效益值 $v(i)$, 任务的结束时间 $D \in \mathbb{Z}^+$, 其中 $\mathbb{Z}^+$ 表示正整数集合. 一个可行调度是对 J 的子集 A 中任务的一种安排, 对于 $i \in A$, $f(i)$ 是开始时间, 且满足下述条件:

$$\begin{cases} f(i) + t(i) \le f(j) \text{或} f(j) + t(j) \le f(i), \text{其中} j \ne i \text{且} i, j \in A \\ \sum\limits_{k \in A} t(k) \le D \end{cases}$$

设机器从 0 时刻开始启动, 只要有作业就不闲置, 求具有最大总效益的调度. 给出算法并分析其时间复杂度.

Solution: 与 0-1 背包问题相类似, 使用 DP 算法, 令 $N_j(d)$ 表示考虑作业集 $\{1, 2, \cdots, j\}$、结束时间为 d 的最优调度的效益, 那么有递推方程

$$N_j(d) = \begin{cases} \max\left\{N_{j-1}(d), N_{j-1}(d - t(j)) + v_j\right\}, & d \ge t(j) \\ N_{j-1}(d), & d < t(j) \end{cases}$$

并且边界 (初始) 条件为

$$N_1(d) = \begin{cases} v_1, & d \ge t(1) \\ 0, & d < t(1) \end{cases}, \quad N_j(0) = 0, \quad N_j(d) = -\infty \left(\text{其中} d < 0\right)$$

自底向上计算, 存储使用备忘录 (以存代算), 可以使用标记函数 $B(j)$ 记录使得 $N_j(d)$ 达到最大时是否

$$N_{j-1}(d - t(j)) + v_j > N_{j-1}(d)$$

如果是, 则 $B(j) = j$; 否则 $B(j) = B(j-1)$.(换句话说, 如果装了作业 j, 那么就追踪其下标; 否则就不追踪更新)

伪代码如后页算法2中所示, 由此我们可以分析出时间复杂度: 得到最大效益 $N[n, D]$ 后, 通过对 $B[n, D]$ 的追踪就可以得到问题的解, 算法的主要工作在于第 7 行到第 16 行的 for 循环, 需要执行 $O(nD)$ 次, 循环体内的工作量是常数时间, 因此算法的总时间复杂度为 $O(nD)$. 显然该算法是**伪多项式时间**的算法[1].

[1]问题就在于如果 D 过大, 即 D 的 2 进制表示会很长, 且 $O(n \cdot D) = O\left(n \cdot 2^{\log(D)}\right) = O\left(n \cdot 2^{\text{输入长度}}\right)$, 是与输入长度相关的指数表达式, 这种复杂度形式的算法称之为**伪多项式时间算法**.

Algorithm 2 Homework 算法

Input: 加工时间 $t[1,\cdots,n]$, 效益 $v[1,\cdots,n]$, 结束时间 D

Output: 最优效益 $N[i,j]$, 标记函数 $B[i,j], i=1,2,\cdots,n, j=1,2,\cdots,D$

for $d=1; d\le t[1]-1; d$++ **do**
    $N[1,d]\leftarrow 0, B[1]\leftarrow 0;$
end for
for $d=t[1]; d\le D; d$++ **do**
    $N[1,d]\leftarrow v[1], B[1]\leftarrow 1;$
end for
for $j=2; j\le n; j$++ **do**
    for $d=1; d\le D; d$++ **do**
        $N[j,d]\leftarrow N[j-1,d];$
        $B[j,d]\leftarrow B[j-1,d];$
        if $d\ge t[j]$ && $N[j-1,d-t[j]]+v[j]>N[j-1,d]$ **then**
            $N[j,d]\leftarrow N[j-1,d-t[j]]+v[j];$
            $B[j,d]\leftarrow j;$
        end if
    end for
end for
end {Homework}

Problem 7

设 A 是顶点为 $1,2,\cdots,n$ 的凸多边形, 可以用不在内部相交的 $n-3$ 条对角线将 A 划分成三角形, 下图中就是 5 边形的所有划分方案. 假设凸 n 边形的边及对角线的长度 d_{ij} 都是给定的正整数, 其中 $1\le i<j\le n$. 划分后三角形 ijk 的权值等于其周长, 求具有最小权值的划分方案. 设计一个动态规划算法求解该问题, 并说明其时间复杂度 (提示: 参考矩阵连乘问题).

如下图1所示, n 边形的顶点是 $1,2,\cdots,n$. 顶点 $i-1,i,\cdots,j$ 构成的凸多边形记作 $A[i,j]$, 于是原始问题就是 $A[2,n]$.

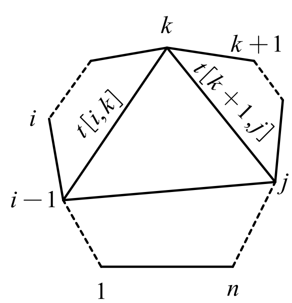

图 1: 子问题归约图

考虑子问题 $A[i,j]$ 的划分, 假设它的所有划分方案中最小权值为 $t[i,j]$. 从 $i,i+1,\cdots,j-1$ 中任选顶点 k, 它与底边 $(i-1)j$ 构成一个三角形 (图1中的三角形). 这个三角形将 $A[i,j]$ 划分成两个凸多边形: $A[i,k]$ 和 $A[k+1,j]$, 从而产生了两个子问题. 这两个凸多边形的划分方案的最小权值分别为 $t[i,k]$

和 $t[k+1, j]$. 根据 DP 思想, $A[i, j]$ 相对于这个顶点 k 的划分方案的最小权值是

$$t[i,k] + t[k+1, j] + d_{(i-1)k} + d_{kj} + d_{(i-1)j}$$

其中 $d_{(i-1)k} + d_{kj} + d_{(i-1)j}$ 是三角形 $(i-1)kj$ 的周长, 于是得到递推关系

$$t[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k \le j-1} \left\{ t[i,k] + t[k+1,j] + d_{(i-1)k} + d_{kj} + d_{(i-1)j} \right\}, & i < j \end{cases}$$

根据上述递推关系可知, 若最优划分 $t[i, j]$ 与 $(i-1)j$ 相连的三角形第三顶是 k, 则这个划分也是 $A[i,k]$ 和 $A[k+1, j]$ 的最优划分, 即最优子结构性质得证. 可以通过标记函数来得到最小权值对应顶点 k 的位置, 于是该划分算法最坏情况下的时间复杂度为 $O\left(n^3\right)$.

```
void MatrixChain(int **d, int n, int **t, int **s) {
    for (int i = 1; i <= n; i++) t[i][i] = 0;
    for (int r = 2; r <= n; r++){
        for (int i = 1; i <= n - r + 1; i++){
            int j = i + r - 1;
            t[i][j] = t[i + 1][j] + d[i - 1][i] + d[i][j] + d[i - 1][j];
            s[i][j] = i;
            for (int k = i + 1; k < j; k++){
                int T = t[i][k] + t[k + 1][j] + d[i - 1][k] + d[k][j] + d[i - 1][j];
                if(t < t[i][j]) {
                    t[i][j] = t, s[i][j] = k;
                }
            }
        }
    }
}
```

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 6&7 课程作业解答

2022 年 11 月 8 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

假设对称旅行商问题的邻接矩阵如下所示, 试用优先队列式分枝限界算法给出最短环游. 画出状态空间树的搜索图, 并说明搜索过程.

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ & \infty & 16 & 4 & 2 \\ & & \infty & 6 & 7 \\ & & & \infty & 12 \\ & & & & \infty \end{pmatrix}$$

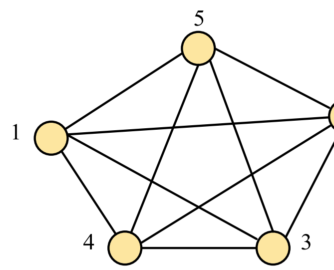

图 1: 旅行商问题

Solution: 状态空间树的搜索图如下图2中所示:

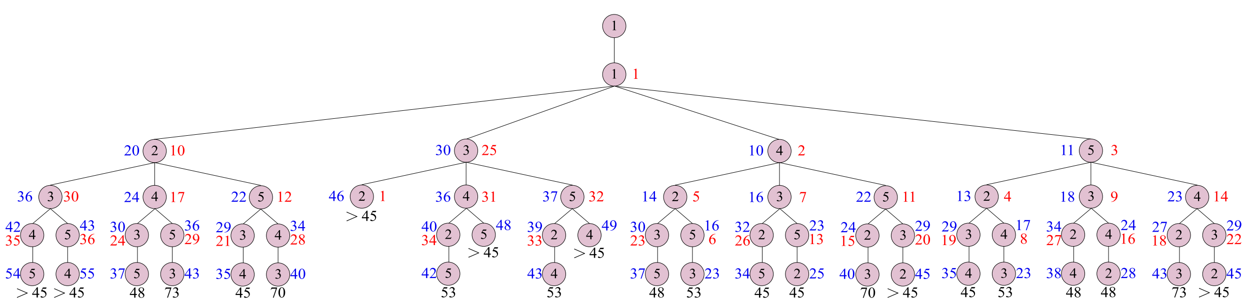

图 2: 解空间树搜索图

图中圆圈内为顶点序号, 蓝色数字为该路径到该节点的总路程, 红色数字表示该节点的搜索序数. 从顶点 1 开始搜索, 将其 4 个儿子节点放入队列. 然后优先访问队列中当前路程最小的节点, 再将它的儿子放入队列. 然后重复上述过程, 优先访问队列中当前路径最小的节点, 将其儿子放入队列. 第一个被访问到的叶子节点为 1-4-2-5-3-1 这条路径, 总路程为 53, 记录当前最短路径. 之后若某节点的路径大于最短路程, 则不再搜索该节点的子树. 若某叶子节点的总路程小于当前最短路径, 则更新之. 如此搜索, 当访问到 1-4-3-5-2-1 这条路径的叶子节点时, 其总路程为 45 < 53, 将 45 更新为当前最短路径, 继续搜索. 最后得出最短环游为 1-2-5-3-4-1、1-4-3-2-5-1、1-4-3-5-2-1、1-5-2-3-4-1, 总路程均为 45.

Problem 2

最佳调度问题: 假设有 n 个任务要由 k 个可并行工作的机器来完成, 完成任务 i 需要的时间为 t_i. 试设计一个分枝限界算法, 找出完成这 n 个任务的最佳调度, 使得完成全部任务的时间 (从机器开始加工任务到最后停机的时间) 最短.

Solution: 限界函数: 将 n 个任务按照所需时间非递减排序, 得到任务序列 $1, 2, \cdots, n$, 满足时间关系 $t[1] < t[2] < \cdots < t[n]$. 将 n 个任务中的前 k 个任务分配给当前 k 个机器, 然后将第 $k+1$ 个任务分配给最早完成已分配任务的机器, 依次进行, 最后找出这些机器最终分配任务所需时间最长的, 此时间作为分支限界函数. 如果一个扩展节点所需的时间超过这个已知的最优值, 则删掉以此节点为根的子树. 否则更新最优值.

优先级: 哪台机器完成当前任务的时间越早, 也就是所有机器中最终停机时间越早, 优先级就越高, 即被选作最小堆中的堆顶, 作为扩展节点. 分支限界算法如下所示:

```
Node{
    int Path[n];
    int T[k];
    int Time;
    int length;
}
Proc BestDispatch(int n, int k, int t[]){
    Node Boot, X, P, result;
    int f;
    f = n * max(t[]);
    Boot.T[n] = {0};
    Boot.Time = 0;
    Boot.Path[n] = {0};
    Boot.length=0;
    AddHeap(Boot);
    while (!Heap.empty()) do {
        P = DeleteMinHeap();
        for i = 1 to k do {
            X = Newnode(P.Path[], P.T[], P.length + 1);
            X.Path[X.length] = i;
            X.T[i] = X.T[i] + t[X.length];
            X.Time = max(X.T[]);
            if X.length == n then {
                if X.Time < f then {
                    f = X.Time;
                    result = X;
                }
            }
            else {
                if X.Time < f then {
                    AddHeap(X);
                }
            }
        }
    }
}
end {BestDispatch}
```

Problem 3

恰好覆盖问题: 设给定有限集 $A = \{a_1, a_2, \cdots, a_n\}$ 和 A 的子集的集合 $W = \{S_1, S_2, \cdots, S_m\}$. 求子集 W 的子集 U, 使得 U 中的子集都不相交且他们的并集等于 A. 求满足条件的所有子集 U.

Solution: 解向量为 $(x_1, x_2, \cdots, x_m)$, $x_i = 0, 1$, 其中 $x_i = 1$ 当且仅当 $S_i \in W$. 部分向量 $(x_1, x_2, \cdots, x_k)$ 表示已经考虑了对 $S_1, S_2, \cdots, S_k$ 的选择. U 为当前所选择集合的并集. 回溯算法如下所示:

Algorithm 1 Eaxtsetcover(U, k)

```
if k = m + 1 then
    return ;
end if
if |U + W(k)| = |U| + |W(k)| then
    X[k] = 1;
    if |S + W(k)| = |A| then
        print X[ ], return ;
    else
        Eaxtsetcover(U + W(k), k + 1);
    end if
end if
X[k] = 0;
Eaxtsetcover(U, k + 1);
end {Eaxtsetcover}
```

Problem 4

分派问题: 给 n 个人分派 n 件工作, 给第 i 人分派第 j 件工作的成本是 $C(i, j)$, 试用分枝限界法求成本最小的工作分配方案.

Solution: 设 n 个人的集合是 $\{1, 2, \cdots, n\}$, n 项工作的集合是 $\{1, 2, \cdots, n\}$, 每个人恰好 1 项工作. 于是有

$$\text{把工作} j \text{分配给} i \Leftrightarrow x_i = j, \quad i, j = 1, 2, \cdots n$$

设解向量为 $X = \langle x_1, x_2, \cdots, x_n \rangle$, 分配成本为 $C(X) = \sum_{i=1}^{n} C(i, x_i)$. 搜索空间是排列树. 部分向量 $\langle x_1, x_2, \cdots, x_k \rangle$ 表示已经考虑了人 $1, 2, \cdots, k$ 的工作分配. 节点分支的约束条件为:

$$x_{k+1} \in \{1, 2, \cdots, n\} \setminus \{x_1, x_2, \cdots, x_k\}$$

可以设立代价函数:

$$F(x_1, x_2, \cdots, x_k) = \sum_{i=1}^{k} C(i, x_i) + \sum_{i=k+1}^{n} \min\{C(i, t) : t \in \{1, 2, \cdots, n\} \setminus \{x_1, x_2, \cdots, x_k\}\}$$

界 B 是已得到的最好可行解的分配成本. 如果代价函数大于界, 则剪枝并回溯. 可用优先队列分支限界算法, 以 F 最小优先扩展. 根节点有 n 个儿子 $x_1 = 1, 2, \cdots n$, 儿子节点有 $n - 1$ 个儿子 $x_2 = 2, 3, \cdots n$; $x_2 = 1, 3, 4, \cdots n; \cdots$. 算法描述如下, 其时间复杂度为 $O(n \cdot n!)$.

```
Node{
    int Path[n];
    int work[n];
    int T[k];
    int Time;
    int length;
}
Proc BestDispatch(int n, int k, int t[]){
    Node Boot, X, P, result;
    int f;
    f = n * max(t[]);
    Boot.T[n] = {0};
    Boot.Time = 0;
    Boot.Path[n] = {0};
    Boot.length=0;
    AddHeap(Boot);
    while (!Heap.empty()) do {
        P = DeleteMinHeap();
        for i = 1 to n do {
            if(work[i] == 0) {
                X = Newnode(P.Path[], P.T[], P.length + 1);
                work[i] = 1;
            }
            X.Path[X.length] = i;
            X.T[i] = X.T[i] + t[X.length];
            X.Time = max(X.T[]);
            if X.length == n then {
                if X.Time < f then {
                    f = X.Time;
                    result = X;
                }
            }
            else {
                if X.Time < f then {
                    AddHeap(X);
                }
            }
        }
    }
}
end {BestDispatch}
```

Problem 5

如图3所示, 一个 4 阶 Latin 方是一个 4×4 的方格, 在它的每个方格内填入 $1, 2, 3$ 或 4, 并使得每个数字在每行、每列都恰好出现一次. 用回溯法求出所有第一行为 1,2,3,4 的所有 4 阶 Latin 方. 将每个解的第 2 行到第 4 行的数字从左到右写成一个序列. 如图3中的解是 $\langle 3, 4, 1, 2, 4, 3, 2, 1, 2, 1, 4, 3 \rangle$. 给出所有可能的 4 阶 Latin 方.

1	2	3	4
3	4	1	2
4	3	2	1
2	1	4	3

图 3: Latin 方

Solution: 解向量为 $X[4][4]$, 约束条件: 同行、同列不能相等. 因此回溯算法设计如下:

```
bool Latincheck(int X[4][4], int k, int row, int line) {
    for(int i = 0; i < row; i++) {
        if(X[i][line] == k) return false;
    }
    for(int j = 0; j < line; j++) {
        if(X[row][j] == k) return false;
    }
    return true;
}
void LatinMatrix(int X[4][4], int row, int line) {
    if(row == 3 && line == 3) {
        for(int i = 0; i < 4; i++)
            for(int j = 0; j < 4; j++)
                cout << X[i][j] << " ";
        return ;
    }
    else {
        for(int color = 1; color <= 4; color++) {
            if(Latincheck(X, color, row, line)) {
                X[row][line] = color;
                if(line < 3) LatinMatrix(X, row, line + 1);
                else if (line == 3 && row != 3) LatinMatrix(X, row + 1, 0);
            }
        }
    }
}
```

University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 8 课程作业解答

2022 年 11 月 27 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

碰撞集问题: 给定一组集合 $\{S_1, S_2, \cdots, S_n\}$ 和预算 b, 问是否存在一个集合 H, 其大小不超过 b, 且 H 和所有 $S_i (i = 1, 2, \cdots, n)$ 相交非空. 请证明碰撞集问题是 $\mathcal{NPC}$ 问题.

Solution:

- **先证明该问题是一个 $\mathcal{NP}$ 问题**: 假设给出集合 H 的所有元素, 显然可以在**多项式时间内**验证该集合 H 是否满足条件要求 (和 S_i 逐一比较是否有交集并检查规模是否超过 b), 所以该问题 $\in \mathcal{P}$ 问题 $\subseteq \mathcal{NP}$ 问题.
- **再利用一个已知的 $\mathcal{NPC}$ 问题, 将其 (在多项式时间内) 归约到目标问题**: 已知**图的顶点覆盖问题 (VC)** 是 $\mathcal{NPC}$ 问题, 只要找到一种把 VC 问题归约到碰撞集问题的多项式方法, 即可证明碰撞集问题是 $\mathcal{NPC}$ 问题. 具体的归约方式构造如下:

  假设有图 $G = (V, E)$, 则把该图的每一条边对应一个集合 S_i, 该边的两点作为该集合的元素, 即每个集合都有两个元素 (如 $S_1 = \{v_1, v_2\}$). 这样就可以构造出 $|E|$ 个集合 $\left\{S_1, S_2, \cdots, S_{|E|}\right\}$, 再将 VC 问题中覆盖的顶点数上限 K 作为预算 b, 并把图 $G = (V, E)$ 的顶点覆盖中的所有点作为集合 H 的元素. 另外还需要说明一下上述多项式变换的充分必要性:

  - **当碰撞集问题有解时, 则顶点覆盖问题就有解:** 只需要选取碰撞集的解 H 对应的所有点, 即为对应顶点覆盖问题的解;
  - **当顶点覆盖问题有解时, 则碰撞集问题就有解:** 当顶点覆盖问题有解 V 时, 则将 V 中的每个顶点对应到所生成的那一组集合 (即 $\left\{S_1, S_2, \cdots, S_{|E|}\right\}$) 中的元素, 从而得到集合 H, 即为碰撞集问题的解.

这样就把 VC 问题归约到碰撞集问题了, 而 VC 问题是 $\mathcal{NPC}$ 问题, 因此碰撞集问题是 $\mathcal{NPC}$ 问题, $\square$.

Problem 2

0-1 整数规划问题: 给定 $m \times n$ 的矩阵 A, n 维整数列向量 c, m 维整数列向量 b 以及整数 D, 问是否存在 n 维 0-1 列向量 x, 使得 $Ax \le b$ 且 $c^{\mathrm{T}} x \ge D$. 请证明 0-1 整数规划问题是 $\mathcal{NPC}$ 问题.

Solution:

- **先证明该问题是一个 $\mathcal{NP}$ 问题**: 对于一个待验证的解, 只需要做一次矩阵乘法和向量内积, 再逐分量的比较即可验证不等式 (即解的正确性). 这个过程是多项式时间内可完成的. 因此 0-1 整数规划问题即为 $\mathcal{NP}$ 问题.
- **再利用一个已知的 $\mathcal{NPC}$ 问题, 将其归约到目标问题**: 为了简化归约过程, 这里我们采用**图的团问题**作为已知的 $\mathcal{NPC}$ 问题[1]. 对于团问题的每一个实例 I: 无向图 $G = (V, E)$ 和非负整数 $J \le |V|$, 其中 $V = \{v_1, v_2, \cdots, v_n\}$. 则在 0-1 整数规划中对应的实例 $f(I)$ 为:

$$\begin{cases} \sum\limits_{i=1}^{n} x_i \ge J \\ x_i + x_j \le 1, \quad \left(v_i, v_j\right) \notin E, i \ne j \\ x_i = 0, 1, \quad i = 1, 2, \cdots, n \end{cases}$$

  显然 f 是多项式时间内可计算的. 现在来证明一下多项式归约 f 的充要性:

[1] VC 到团的多项式归约变换 f: 对 VC 的每一个实例 I, 无向图 $G = (V, E)$ 和非负整数 $K \le |V|$. 而团对应的实例 $f(I)$: 无向图 $\overline{G}\left(V, \overline{E}\right)$ 和 $J = |V| - K$. 由于 VC 是 $\mathcal{NPC}$ 问题, 因此团就是 $\mathcal{NPC}$ 问题.

– 设 $V' \subseteq V$ 是 G 的一个团且 $|V'| \geq J$, 令

$$x_i = \begin{cases} 1, & \text{若} v_i \in V' \\ 0, & \text{否则} \end{cases}$$

于是当 $(v_i, v_j) \notin E$ 时, 则有 $x_i + x_j \leq 1$ 且 $\sum_{i=1}^{n} x_i = \left|V'\right| \geq J$;

– 反之, 取 $V' = \{v_i | x_i = 1\}$, 则 $\forall v_i, v_j \in V', x_i + x_j = 2$, 从而 $(v_i, v_j) \in E$.

又因为 $\left|V'\right| = \sum_{i=1}^{n} x_i \geq J$. 即 $V'(\subseteq V)$ 就是 G 的一个顶点数不小于 J 的团.

至此, 图的团问题就归约到 0-1 整数规划问题了, 由于图的团问题是 $\mathcal{NPC}$ 问题, 因此 0-1 整数规划问题也是 $\mathcal{NPC}$ 问题, □.

Problem 3

独立集问题: 任给一个无向图 $G = (V, E)$ 和非负整数 $J \leq |V|$, 问 G 中是否存在顶点数不小于 J 的独立集. 请证明独立集问题是 $\mathcal{NPC}$ 问题.

Solution:

- **先证明该问题是一个 $\mathcal{NP}$ 问题**: 独立集的验证以及计算其顶点数显然可以在多项式时间内完成, 故独立集问题显然是 $\mathcal{NP}$ 问题;

- **再利用一个已知的 $\mathcal{NPC}$ 问题, 将其归约到目标问题**: 为了归约过程简单, 这里我们选取**图的顶点覆盖问题**作为参照物[2]. 归约过程为: 任给顶点覆盖问题的一个实例, 它是由无向图 $G = (V, E)$ 和非负整数 $K \leq |V|$ 组成, 对应独立集问题的实例由无向图 $G = (V, E)$ 和非负整数 $J = |V| - K$ 组成. 图 $G = (V, E)$ 有 K 顶点覆盖 $V' \Leftrightarrow$ 图 G 有 $n - k$ 个独立集 $V \backslash V'$, **因此该变换是充要的**. 故而图的顶点覆盖问题就归约到独立集问题了, 由于顶点覆盖问题是 $\mathcal{NPC}$ 问题, 因此独立集问题问题也是 $\mathcal{NPC}$ 问题, □.

Problem 4

已知无向图的 Hamilton 圈问题 (HC) 是 $\mathcal{NPC}$ 问题, 证明 TSP 判定问题是 $\mathcal{NPC}$ 问题.

Solution: 任给一个无向图 $G = (V, E)$ 的一个顶点排序 $v_1 v_2 \cdots v_n$ 和数 L, 验证 $(v_i, v_{i+1}) \in E$ 和 $(v_n, v_1) \in E$ 可在 $O(n^3)$ 时间内完成, 而验证路径只需要 $O(n)$ 时间. 因此 TSP 判定问题 $\in \mathcal{NP}$. 我们考虑以 HC 问题为起点, 将其归约到 TSP 判定问题. 构造 HC 到 TSP 的多项式变换 f: 对 HC 的每一个实例 I, I 是一个无向图 $G = (V, E)$, TSP 对应的实例 $f(I)$ 为: 城市集合 V, 任意两个不同的城市 u 和 v 之间的距离为

$$d(u, v) = \begin{cases} 1, & \text{若} (u, v) \in E \\ 2, & \text{若} (u, v) \notin E \end{cases}$$

以及界限 $D = |V|$. 显然 f 是多项式时间内可计算的, 又因为 $f(I)$ 中每一个城市恰好经过一次的巡回路线有 $|V|$ 条边, 每条边的长度为 1 或 2. 故而巡回路线的长度至少为 D. 因此巡回路线的长度不超过 D 当且仅当巡回路线的长度为 D, 当且仅当它的每条边长度都为 1, 当且仅当它是 G 中的一条哈密顿回路. 综上所述, $I \in Y_{\mathrm{HC}} \Leftrightarrow f(I) \in Y_{\mathrm{TSP}}$, 即多项式规约变换 f 的充要性是满足的. 又因为 HC 问题是 $\mathcal{NPC}$ 问题, 故 TSP 判定问题也是 $\mathcal{NPC}$ 问题.

[2]不用像参考答案 (以团问题为参照物来归约到独立集问题) 那样复杂.

Problem 5

0-1 背包 (优化) 问题: 有 n 个物品, 他们各自的体积 w_i 和价值 p_i, 现有给定容量 M 的背包, 如何将背包装入的物品具有最大价值总和? 请说明 0-1 背包问题是 $\mathcal{NP}$ 困难问题, 但不是 $\mathcal{NPC}$ 问题.

Solution: 要验证一个可行解是否为下述问题的最优解

$$\begin{cases} \max \sum_{i=1}^{n} x_i p_i \\ \sum_{i=1}^{n} x_i w_i \le M \\ x_i = 0, 1 \ (i = 1, \cdots, n) \end{cases}$$

则需要**比较所有的可行解** $X = (x_1, x_2, \cdots, x_n), x_i \in \{0, 1\}$. 这最多有 2^n 种可能, 而基于比较的求最大值问题的复杂度下界为 $\Omega(M)$, 因此本问题对于"最优解"的正确性验证需要消耗的时间下界为 $O(2^n)$. 即不存在多项式时间算法, 使得其能够验证解的正确性. 也就是说, 0-1 背包 (优化) 问题不是 $\mathcal{NP}$ 的, 所以肯定就不是 $\mathcal{NPC}$ 的.

与此同时, **0-1 背包优化问题不会比 0-1 背包判定问题更容易**, 然而 0-1 背包判定问题是 $\mathcal{NPC}$ 的. 因此 0-1 背包优化问题是 $\mathcal{NPH}$ 的, 但不是 $\mathcal{NPC}$ 的.

Problem 6

$\mathcal{NPC}$ 问题一定是 $\mathcal{NPH}$ 问题么?

Solution: $\mathcal{NPC}$ 问题一定是 $\mathcal{NPH}$ 问题. $\mathcal{P}$,$\mathcal{NP}$,$\mathcal{NPC}$,$\mathcal{NPH}$ 问题的关系如下图1所示:

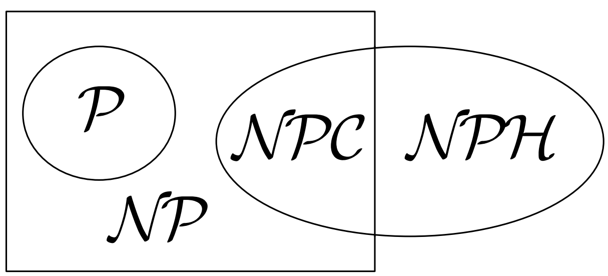

图 1: P-NP-NPC-NPH 问题关系图

Problem 7

对于给定 $x \neq 0$, 求 n 次多项式 $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ 的值.

- 设计一个最坏情况下时间复杂度为 $\Theta(n)$ 的求值算法;
- 证明任何求值算法的时间复杂度都是 $\Omega(n)$.

Solution: (1). 采用秦九韶算法, 递归计算如下:

$$
\begin{aligned}
P_0(x) &= a_n \\
P_1(x) &= P_0(x)\cdot x + a_{n-1} \\
P_2(x) &= P_1(x)\cdot x + a_{n-2} \\
P_3(x) &= P_2(x)\cdot x + a_{n-3} \\
&\vdots \\
P_n(x) &= P_{n-1}(x)\cdot x + a_{n-3}
\end{aligned}
$$

上述算法用一个 for 循环就可以实现, 并且循环体内部的操作就只有加法和乘法, 即循环体消耗常数时间并且循环次数为 $n+1$. 因此算法在最坏情形下的时间复杂度为 $\Theta(n)$.

(2). 多项式 $P(x)$ 有 $n+1$ 个系数, 对于任意一个算法 $\mathcal{A}$, 都需要对**每一个系数**至少做一次处理, 否则算法就可能得到错误的输出. 即任何正确的算法都至少需要 $n+1$ 次运算, 即最坏情况下的时间复杂度下界为 $\Omega(n)$.

Problem 8

写出下述优化问题对应的判定问题, 并证明这些判定问题 $\in \mathcal{NP}$:

- **最长回路优化问题**: 任给无向图 G, 在 G 中找到一条最长的初级 (即顶点不重复的) 回路.
- **图着色优化问题**: 任给无向图 $G=(V,E)$, 给 G 的每一个顶点涂一种颜色, 要求任一条边的两个端点的颜色都不相同. 如何用最少的颜色给 G 的顶点着色? 即求映射 $f: V \to \mathbb{Z}^+$ 满足条件 $\forall (u,v)\in E, f(u)\neq f(v)$, 且使 $\mathrm{card}\{f(u)\,|u\in V\}$ 最小.

Solution: 上述优化问题对应的判定问题为:

- **最长回路判定问题**: 任给无向图 $G=(V,E)$ 和正整数 $L\leq |V|$, 在 G 中能否找到一条长度 $\geq L$ 的初级 (即顶点不重复的) 回路?
- **最长回路判定问题**的非确定性多项式时间算法 $\mathcal{A}$: 猜想对 G 的任意一条初级回路 C, 若 C 的长度 $\geq L$, 则回答"yes", 否则回答"no". 显然该问题 $\in \mathcal{NP}$.
- **图着色判定问题**: 任给无向图 $G=(V,E)$ 和正整数 K, 给 G 的每一个顶点涂一种颜色, 要求任一条边的两个端点的颜色都不相同. 能否用不超过 K 种颜色给 G 的顶点着色? 即是否存在映射 $f: V\to\mathbb{Z}^+$ 满足条件 $\forall (u,v)\in E, f(u)\neq f(v)$, 且使 $\mathrm{card}\{f(u)\,|u\in V\}\leq K$?
- **图着色判定问题**的非确定性多项式时间算法 $\mathcal{B}$: 猜想用不超过 K 种颜色给 G 的每一个顶点涂色, 检查每一条边的两个端点的颜色是否都不相同. 若是, 则回答"yes", 否则回答"no". 即猜想一个单射 $f: V\to\{1,2,\cdots,K\}$, 若满足条件 $\forall (u,v)\in E, f(u)\neq f(v)$, 则回答"yes", 否则回答"no". 于是可知该问题 $\in \mathcal{NP}$.

计算机算法设计与分析

083500M01001H

Chap 9&10&11 课程作业

2022 年 12 月 1 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

判断正误:

- Las Vegas 算法不会得到不正确的解. ( )
- Monte Carlo 算法不会得到不正确的解. ( )
- Las Vegas 算法总能求得一个解. ( )
- Monte Carlo 算法总能求得一个解. ( )

Solution:

- 正确, 拉斯维加斯算法不会得到不正确的解. 一旦用拉斯维加斯算法找到一个解, 这个解就一定是正确解. 但有时用拉斯维加斯算法找不到解.
- 错误, Monte Carlo 算法每次都能得到问题的解, 但不保证所得解的正确性. 请注意, 可以在 Monte Carlo 算法给出的解上加一个验证算法, 如果正确就得到解, 如果错误就不能生成问题的解, 这样 Monte Carlo 算法便转化为了 Las Vegas 算法.
- 错误, Las Vegas 算法并不能保证每次都能得到一个解, 但是如果一旦某一次得到解, 那么就一定是正确的.
- 正确, Monte Carlo 算法每次运行都能给出一个解, 但正确性就不能保证了.

Problem 2

判断正误:

- 一般情况下, 无法有效判定 Las Vegas 算法所得解是否正确. ( )
- 一般情况下, 无法有效判定 Monte Carlo 算法所得解是否正确. ( )
- 虽然在某些步骤引入随机选择, 但 Sherwood 算法总能求得问题的一个解, 且所求得的解总是正确的. ( )
- 虽然在某些步骤引入随机选择, 但 Sherwood 算法总能求得问题的一个解, 但一般情况下, 无法有效判定所求得的解是否正确. ( )

Solution:

- 错误, Las Vegas 算法并不能保证每次都能得到解, 但是如果一旦某一次得到解, 那么就一定是正确的.
- 错误, 虽然 Monte Carlo 算法每次运行都能给出一个解, 可能是错的也可能是对的, 但是可以通过检验解来有效判定其正确性. 判定解的正确性跟算法本身没有多大关系, 只要代进去验证即可. 特殊点在于, 只要 Las Vegas 算法求得解了, 那么就一定是正确的, 就不用再浪费时间来判定了; 但是对于 Monte Carlo 算法的所得解, 必须要进行正确性检验.
- 正确, Sherwood 算法总能求得问题的一个解, 且所求得的解总是正确的.
- 错误.

Problem 3

判断正误:

- 旅行商问题存在多项式时间近似方案. ( )
- 0/1 背包问题存在多项式时间近似方案. ( )
- 0/1 背包问题的贪心算法 (单位价值高优先装入) 是绝对近似算法. ( )
- 多机调度问题的贪心近似算法 (按输入顺序将作业分配给当前最小负载机器) 是 ϵ-近似算法. ( )

Solution:

- 错误. 根据教材可知, 旅行商问题不存在多项式时间近似算法, 除非 $\mathcal{P} = \mathcal{NP}$. 如果存在的话, 那么就可以证得 $\mathcal{P} = \mathcal{NP}$, 即可以拿图灵奖了.
- 正确, **PTAS 算法**就是 0/1 背包问题的多项式时间近似方案.
- 错误, 0/1 背包问题的贪心算法**不是**绝对近似算法.
- 正确, 多机调度问题的贪心近似算法有 GMPS 和 DGMPS 分别是 2-近似和 3/2-近似算法.

Problem 4

设 Las Vegas 算法获得解的概率为 $p(x) \geq \delta, 0 < \delta < 1$, 则调用 k 次算法后, 获得解的概率为:_____.
Solution: 不妨求一下调用 k 次算法后, 求解失败 (即 k 次调用都求解失败) 的概率:

$$P\left(\text{失败}\right) = (1 - p(x))^k \leq (1-\delta)^k \Rightarrow P\left(\text{成功}\right) = 1 - P\left(\text{失败}\right) \geq 1 - (1-\delta)^k$$

即获得解的概率至少为 $1 - (1-\delta)^k \to 1$(当$k \to \infty$).

Problem 5

对于判定问题 Π 的 Monte Carlo 算法, 当返回 false(true) 时解总是正确的, 但当返回 true(false) 时解可能有错误, 该算法是 ______.

(A) .偏真的*Monte Carlo*算法 (B) .偏假的*Monte Carlo*算法
(C) .一致的*Monte Carlo*算法 (D) .不一致的*Monte Carlo*算法

Solution: 答案选 B, 只要将偏真的 Monte Carlo 算法的定义中的 true/false 互换即可得到偏真的 Monte Carlo 算法的定义.

Problem 6

写出禁忌搜索算法的主要步骤.

Solution: 禁忌搜索算法的主要步骤如下算法1中所示:

Algorithm 1 禁忌搜索算法步骤

选定一个初始可行解 $\boldsymbol{x}^{\text{cb}}$ 并初始化禁忌表 $H \leftarrow \{\}$;
while 不满足停止规则 **do**
  在 $\boldsymbol{x}^{\text{cb}}$ 的邻域中选出满足禁忌要求的候选集 Can-$N\left(\boldsymbol{x}^{\text{cb}}\right)$;
  从该候选集中选出一个评价值最佳的解 $\boldsymbol{x}^{\text{lb}}$;
  令 $\boldsymbol{x}^{\text{cb}} \leftarrow \boldsymbol{x}^{\text{lb}}$ 并更新记录 H;
end while

Problem 7

禁忌对象特赦可以基于影响力规则: 即特赦影响力大的禁忌对象. 影响力大什么含义? 举例说明该规则的好处.

Solution: 影响力大意味着有些对象变化对目标值影响很大. 如 0/1 背包问题, 当包中无法装入新物品时, 特赦体积大的分量来避开局部最优解.

Problem 8

判断正误:

- 禁忌搜索中, 禁忌某些对象是为了避免领域中的不可行解. ( )
- 禁忌长度越大越好. ( )
- 禁忌长度越小越好. ( )

Solution:

- 错误, 选取禁忌对象是为了引起解的变化, 根本目的在于避开邻域内的局部最优解而不是不可行解.
- 错误, 禁忌长度短了则可能陷入局部最优解.
- 错误, 禁忌长度长了则导致计算时间长.

Problem 9

写出模拟退火算法的主要步骤.

Solution: 模拟退火算法的主要步骤如下算法2中所示:

Algorithm 2 模拟退火算法步骤

任选初始解 x_0 并初始化 $x_i \leftarrow x_0, k \leftarrow 0, t_0 \leftarrow t_{\max}$(初始温度);
while $k \le k_{\max}$ && $t_k \ge T_f$ **do**
    从邻域 $N(x_i)$ 中随机选择 x_j, 即 $x_j \leftarrow_R N(x_i)$;
    计算 $\Delta f_{ij} = f(x_j) - f(x_i)$;
    if $\Delta f_{ij} \le 0 \,\|\, \exp\left(-\Delta f_{ij}/t_k\right) > \mathrm{RANDOM}(0,1)$ **then**
        $x_i \leftarrow x_j$;
    end if
    $t_{k+1} \leftarrow d(t_k)$;
    $k \leftarrow k + 1$;
end while

Problem 10

写出遗传算法的主要步骤.

Solution: 遗传算法的主要步骤如下算法3中所示:

Algorithm 3 遗传算法步骤

选择问题的一个编码并初始化种群 (N 个染色体)pop (1) $:= \left\{\mathrm{pop}_j\,(1)\,|\,j = 1,2,\cdots,N\right\}, t := 1$;
对种群 pop(1) 的每个染色体 $\mathrm{pop}_i(1)$ 计算其适应性函数 $f_i = \mathrm{fitness}(\mathrm{pop}_i(1))$;
while 停止规则不满足 **do**
    计算得出概率分布 $p_i = \dfrac{f_i}{\sum_{1\le j\le N} f_j}$ $(*)$;
    根据概率分布 $(*)$ 从 $\mathrm{pop}(t)$ 中随机选取 N 个染色体并形成种群

$$\mathrm{newpop}(t+1) := \left\{\mathrm{pop}_j\,(t)\,|\,j = 1,2,\cdots,N\right\}$$

    通过交叉 (交叉概率为 P_c) 得到一个有 N 个染色体的种群 $\mathrm{crosspop}(t+1)$;
    以较小的概率 p, 使得染色体的基因发生变异, 形成种群 $\mathrm{mutpop}(t+1)$;
    $t := t + 1$, 诞生新种群 $\mathrm{pop}(t) := \mathrm{mutpop}(t)$;
    对种群 $\mathrm{pop}(t)$ 的每个染色体 $\mathrm{pop}_i(t)$ 计算其适应性函数 $f_i = \mathrm{fitness}(\mathrm{pop}_i(t))$;
end while

Problem 11

为避免陷入局部最优 (小), 模拟退火算法以概率 $\exp\left(-\Delta f_{ij}/t_k\right)$ 接受一个退步 (比当前最优解差) 的解, 以跳出局部最优. 试说明参数 $t_k, \Delta f_{ij}$ 对是否接受退步解的影响.

Solution: 很明显, 当 t_k 较大时, 接受退步解的概率越大; 当 Δf_{ij} 较大时, 接受退步解的概率越小.

Problem 12

下面属于模拟退火算法实现的关键技术问题的有 ______.

(A) .初始温度 (B) .温度下降控制 (C) .邻域定义 (D) .目标函数

Solution: 模拟退火算法实现的关键技术问题有**邻域的定义 (构造)、起始温度的选择、温度下降方法、每一温度的迭代长度以及算法终止规则**. 因此选择 (A), (B), (C).

Problem 13

用遗传算法解某些问题, fitnees $= f(x)$ 可能导致适应函数难以区分这些染色体. 请给出一种解决办法.

Solution: 采用线性加速适应函数: fitness $(x) = \alpha f(x) + \beta f(x)$

Problem 14

用非常规编码染色体实现的遗传算法, 如 TSP 问题使用 $1, 2, \cdots, n$ 的排列编码, 简单交配会产生什么问题? 如何解决?

Solution: 后代可能会出现非可行解, 因此需要通过罚值和交叉新规则来解决.

Problem 15

下面属于遗传算法实现的关键技术问题的有 ______.

(A) .解的编码　　(B) .初始种群的选择　　(C) .邻域定义　　(D) .适应函数

Solution: 遗传算法实现的关键技术问题有**解的编码、适应函数、初始种群的选取、交叉规则以及终止规则**. 因此选择 (A), (B), (D).

Problem 16

设旅行商问题的解表示为 $D = F = \left\{S|S = (i_1, i_2, \cdots, i_n), i_1, i_2, \cdots, i_n\text{是}1, 2, \cdots, n\text{的一个排列}\right\}$, 邻域定义为 2-OPT(即 S 中的两个元素对换), 求 $S = (3, 1, 2, 4)$ 的邻域 $N(S)$.

Solution: 将 S 中的两个元素对换即可得到 $N(S)$:

$$N(S) = \{(1,3,2,4), (2,1,3,4), (4,1,2,3), (3,2,1,4), (3,4,2,1), (3,1,4,2)\}$$

Problem 17

0/1 背包问题的解记作 $X = (x_1, x_2, \cdots, x_n), x_i \in \{0, 1\}, i = 1, 2, \cdots, n$. 邻域定义为

$$N(X) = \left\{Y \,\middle|\, \sum_{i=1}^{n} |y_i - x_i| \leq 1\right\}, X = (1,1,0,0,1)$$

求邻域 $N(X)$.

Solution: 每次只允许一个分量变化即可求出邻域 $N(X)$:

$$N(X) = \{(0,1,0,0,1), (1,0,0,0,1), (1,1,1,0,1), (1,1,0,1,1), (1,1,0,0,0)\}$$

Problem 18

顶点覆盖问题: 任给一个图 $G = \langle V, E \rangle$, 求 G 的顶点数最少的顶点覆盖. 复习顶点覆盖问题的近似算法及其证明.

Solution: **MVC**算法如下所示:

Algorithm 4 算法 **MVC**(G)

Input: 图 $G = \langle V, E \rangle$
Output: 最小顶点覆盖 V'

```
V' ← ∅, e_1 ← E;
while e_1 ≠ ∅ do
    从 e_1 中任选一条边 (u, v);
    V' ← V' ∪ {u, v};
    从 e_1 中删去与 u 和 v 相关联的所有边;
end while
return V';
end {MVC};
```

显然算法 MVC 的时间复杂度为 $O(m), m = |E|$. 记 $|V'| = 2k$, V' 中的顶点是 k 条边的端点, 这 k 条边互不关联. 为了覆盖这 k 条边则需要 k 个顶点, 从而 $\mathrm{OPT}(I) \geq k$. 于是有

$$\frac{\mathrm{MVC}\,(I)}{\mathrm{OPT}\,(I)} \leq \frac{2k}{k} = 2$$

故 MVC 是最小顶点覆盖问题的 2-近似算法,□.

Problem 19

1.下面说法，正确的是：____________.

(1)P 类问题是存在多项式时间算法的问题。

(2)NP 类问题是不存在多项式时间算法的问题。

(3)P 类问题一定也是 NP 类问题。

(4)NP 类问题比 P 类问题难解。

2.下面说法，正确的是：__________.

(1)P$\subset$NP (2)P$\subseteq$NP (3)P=NP (4)P$\neq$NP

3. 下面说法，正确的是：_________.

(1)NP-难问题是 NP 中最难的问题

(2)NP-完全问题是 NP 中最难的问题

(3)NP-难不比任何 NP 问题容易

(4)NP-完全问题也是 NP-难问题。

参考答案

1.(1)(3) 2.(2) 3.(2)(3)(4)