苏州大学实验报告

院、系	计算机学院	年级专业	19 计算机类	姓名	张昊	学号	1927405160
课程名称	数据结构课程实践					成绩	
指导教师	孔芳	同组实验者	无		实验日期	2020 年 9 月 18 日	

实 验 名 称 第 1 章习题 算法设计

一、实验目的

通过本次实验要达到如下目的：

1. 了解程序设计的一般过程；
2. 理解算法和数据结构在程序中的作用；
3. 熟练掌握数据结构的概念；
4. 掌握算法的描述方法和时间复杂度的分析方法。

二、实验内容

1. 找出整型数组 A[n]中的最大值和次最大值。
2. 判断给定字符串是否是回文。所谓回文是正读和反读均相同的字符串，例如"abcba"或"abba"是回文，而"abcda"不是回文。
3. 已知数组 A[n]中的元素为整型，设计算法将其调整为左右两部分，左边所有元素为奇数，右边所有元素为偶数，并要求算法的时间复杂度为 O(*n*)。
4. 荷兰国旗问题。要求重新排列一个由字符 R，W， B（R 代表红色，W 代表白色，B 代表蓝色，这都是荷兰国旗的颜色）构成的数组，使得所有的 R 都排在最前面，W 排在其次，B 排在最后。为荷兰国旗问题设计一个算法，其时间性能是 O(*n*)。
5. 有 4 个人打算过桥，这个桥每次最多只能有两个人同时通过。他们都在桥的某一端，并且是在晚上，过桥需要一只手电筒，而他们只有一只手电筒。这就意味着两个人过桥后必须有一个人将手电筒带回来。每个人走路的速度是不同的：甲过桥要用 1 分钟，乙过桥要用 2 分钟，丙过桥要用 5 分钟，丁过桥要用 10 分钟，两个人一起走路的速度等于其中较慢那个人的速度。问题是他们全部过桥最少要用多长时间？

三、实验步骤和结果

1. 找出整型数组 A[n]中的最大值和次最大值。

【算法设计】

最大值可以通过遍历查找发现。具体的，两次遍历数组 A[n]。第一次用临时变量存储第 1 号位置的数组元素，在遍历中如果**发现比临时变量大**的元素就将其赋值给临时变量，遍历结束即可找到最大值。第二次用一个临时变量存储第 1 号位置的数组元素，在遍历中如果发现**比临时变量大但比最大值小**的元素就将其赋值给这个临时变量，这轮遍历结束即可找到次最大值。

【时间复杂度】

n 为数组的规模，两次遍历中比较操作的执行次数为 *2n*，故算法的时间复杂度为 O(*n*)。

【伪代码】

```
Algorithm: findMaxAndSecondaryNumber
Input:  整型数组 A[n]
Output:  最大值和次最大值
    1.  max ← A[1];
    2.  i 循环 2 到 n
        2.1  if A[i] > temp then max ← A[i];
    3.  secondary ← A[1];
```

```
        4.  i 循环 2 到 n
            4.1  if A[i] < temp and A[i] > secondary then secondary ← A[i];
        5.  return max, secondary;
```

【C++语言描述】

注：通过引用参数返回两个值 maxNumber 和 secondaryNumber。

```
void findMaxAndSecondaryNumber(
                const int A[], unsigned size, int &maxNumber, int &secondaryNumber) {
    maxNumber = A[0];
    for (int i = 1; i < size; ++i) {            // Find the max number of the array A[size].
        if (A[i] > maxNumber)
            maxNumber = A[i];
    }
    secondaryNumber = A[0];
    for (int i = 1; i < size; ++i) {
        // Find the secondary max number of the array A[size].
        if (A[i] < maxNumber && A[i] > secondaryNumber)
            secondaryNumber = A[i];
    }
}
```

2. 判断给定字符串是否是回文。

【算法设计】

由回文字符串的定义，**字符串的首尾字符相同。**可以考虑从**字符串起始位置开始与字符串末尾位置字符逐一比较**，直到**首次出现不同的字符**或者**到达字符串正中间位置**停止。若所有比较都相等，则给定字符串是回文，否则不是回文。

【时间复杂度】

n 为字符串的长度，循环中比较操作的执行次数为 *n/2*，故算法的时间复杂度为 O(*n*)。

【伪代码】

```
Algorithm: isPalindromeString
Input:  字符串 str
Output:  是否为回文
    1.  i 循环 1 到 length/2
        1.1  if str[i] ≠ str[length - i] then return FALSE;
    2.  return TRUE;
```

【C++语言描述】

```
bool isPalindromeString(const std::string &str) {
    for (int i = 0; i < str.length() / 2; ++i) {
        if (str[i] != str[str.length() - 1 - i])
            return false;
    }
    return true;
}
```

3. 已知数组 A[n]中的元素为整型，设计算法将其调整为左右两部分，左边所有元素为奇数，右边所有元素为偶数，并要求算法的时间复杂度为 O(*n*)。

【算法设计】

这是一个对数组元素分类的问题。可以用两个数组来分别存放不同类别的元素，但合并两个数组会导致不必要的元素拷贝。那么可以考虑对数组进行原地操作。**由普通遍历的一个**

位置指针出发，考虑使用两个指针，一个指向数组头部，一个指向数组尾部。具体的，当数组**头部指针遇到偶数，尾部指针遇到奇数时交换两个指针指向的元素**；尾部指针遇到偶数，只移动尾部指针；头部指针遇到奇数，只移动头部指针。如此循环直到**头部指针超过或触及的尾部指针**为之。这样即可实现左边所有元素为奇数，右边所有元素为偶数。

例如　现有数组：（头部指针记为 left，尾部指针记为 right）

数组元素	-44	86	-56	23	-11	95	-7	-57	-78	86	36	-33
位置指针	left											right

此时 left 指向-44 为偶数，right 指向-33 为奇数，交换两者并移动指针得到：

数组元素	-33	86	-56	23	-11	95	-7	-57	-78	86	36	-44
位置指针		left									right	

此时 right 指向 36 为偶数，只移动 right 得到：

数组元素	-33	86	-56	23	-11	95	-7	-57	-78	86	36	-44
位置指针		left								right		

同理，多次操作得到：

数组元素	-33	86	-56	23	-11	95	-7	-57	-78	86	36	-44
位置指针		left						right				

此时 left 指向 86 为偶数，right 指向-57 为奇数，交换两者并移动指针得到：

数组元素	-33	-57	-56	23	-11	95	-7	86	-78	86	36	-44
位置指针			left				right					

此时 left 指向-56 为偶数，right 指向-7 为奇数，交换两者并移动指针得到：

数组元素	-33	-57	-7	23	-11	95	-56	86	-78	86	36	-44
位置指针				left		right						

此时 left 为奇数，只移动 left 得到：

数组元素	-33	-57	-7	23	-11	95	-56	86	-78	86	36	-44
位置指针					left	right						

同理，多次操作得到：

数组元素	-33	-57	-7	23	-11	95	-56	86	-78	86	36	-44
位置指针						left right						

此时 left 和 right 重合，循环结束，得到调整后的数组。

【时间复杂度】

n 为数组规模。最好情况下数组不需要交换元素，时间复杂度为 O(1)；最坏情况下为数组左边为偶数，右边为奇数，最多需要交换 *n/2* 对元素，故算法的时间复杂度为 O(*n*)。

【伪代码】

```
Algorithm: adjustArray
Input: 整型数组 A[n]
Output: 无（在数组原地操作）
    1.  头部指针 ← 1, 尾部指针 ← n
    2.  while 头部指针 < 尾部指针
        2.1  if A[头部指针] % 2 ≠ 0 then 头部指针 ← 头部指针 + 1;
        2.2  else if A[尾部指针] % 2 = 0 then 尾部指针 ← 尾部指针 - 1;
        2.3  else then
                2.3.1   头部指针 ↔ 尾部指针;
                2.3.2   头部指针 ← 头部指针 + 1;
                2.3.3   尾部指针 ← 尾部指针 - 1;
```

【C++语言描述】

```
void adjustArray(int a[], unsigned size) {
    int leftPosition = 0, rightPosition = size - 1;
    while (leftPosition < rightPosition) {
        if (a[leftPosition] % 2 != 0) {         // left position element is odd -> pass.
            leftPosition++;
            continue;
        } else if (a[rightPosition] % 2 == 0) {       // right position element is even -> pass.
            rightPosition--;
            continue;
        }
        // left position element is even, right position element is odd -> swap.
        std::swap(a[leftPosition], a[rightPosition]);
        leftPosition++;
        rightPosition--;
    }
}
```

4. 荷兰国旗问题。要求算法的时间复杂度为 O(*n*)。

【算法设计】

由第 3 题的思路启发，可以将这个问题视为一个**数组排序问题**，这个数组分为前中后部三部分，每一个元素 R，W，B 必为其中之一，但数量不均等，故三个部分不等分。只操作 R 和 B，将 R 全排在数组的前部，B 全排在数组的后部，这样中部的 W 自然排好。

具体来说设置**三个位置指针**，分别指向这个数组的开始、结尾和当前位置（从开头开始），使用**当前位置指针**循环遍历：遇到 R 属于前部，**当前位置元素和开始位置指针指向的元素进行交换，然后两者均前进**；遇到 W 属于中部，不动，仅使**当前位置指针前进**；遇到 B 属于后部，**和结尾位置指针指向元素交换**，这里要注意交换后的当前位置元素不一定为 W，还有可能是属于前部的 R，所以**只使结尾位置指针后退**，而保持当前位置指针不变，便于下一次循环的判断。如此循环遍历，直到**当前位置指针超过或触及了结尾位置指针**。

例如　有数组 B W R W R B W B W R（开始位置指针记为 less，结束位置指针记为 more，当前位置指针记为 index）

数组元素	B	W	R	W	R	B	W	B	W	R
位置指针	less index									more

此时 index 遇到 B，和 more 指向元素交换，仅 more 移动位置，得到：

数组元素	R	W	R	W	R	B	W	B	W	B
位置指针	less index								more	

此时 index 遇到 R，和 less 指向元素交换，然后两者均前进得到：

数组元素	R	W	R	W	R	B	W	B	W	B
位置指针		less index							more	

此时 index 遇到 W，仅 index 向前移动得到：

数组元素	R	W	R	W	R	B	W	B	W	B
位置指针		less	index						more	

此时 index 遇到 R，和 less 指向元素交换，然后两者均前进得到：

数组元素	R	R	W	W	R	B	W	B	W	B
位置指针			less	index					more	

此时 index 遇到 W，仅 index 向前移动得到：

数组元素	R	R	W	W	R	B	W	B	W	B
位置指针			less		index				more	

此时 index 遇到 R，和 less 指向元素交换，然后两者均前进得到：

数组元素	R	R	R	W	W	B	W	B	W	B
位置指针				less		index			more	

此时 index 遇到 B，和 more 指向元素交换，仅 more 移动位置，得到：

数组元素	R	R	R	W	W	W	W	B	B	B
位置指针				less		index		more		

此时 index 遇到 W，仅 index 向前移动得到（两步相同的操作后）：

数组元素	R	R	R	W	W	W	W	B	B	B
位置指针				less				index more		

此时 index 和 more 重合，循环结束，得到调整后的数组。

【时间复杂度】

n 为字符数组的长度，最坏情况下当前位置指针需要遍历一遍数组，每次都发生交换，故算法的时间复杂度为 O(*n*)。

【伪代码】

```
Algorithm: sortFlag
Input: 字符数组 F[n]
Output: 无（在数组原地操作）
    1.  开始位置指针 ← 1，结束位置指针 ← n，当前位置指针 ← 1
    2.  while 当前位置指针 < 结束位置指针
        2.1 if F[当前位置指针] = 'R' then
                2.1.1   F[当前位置指针] ↔ F[开始位置指针];
                2.1.2   当前位置指针 ← 当前位置指针 + 1;
                2.1.3   开始位置指针 ← 开始位置指针 + 1;
        2.2 else if F[当前位置指针] = 'W' then 当前位置指针 ← 当前位置指针 + 1;
        2.3 else if F[当前位置指针] = 'B' then
                2.3.1   F[当前位置指针] ↔ F[结束位置指针];
                2.3.2   结束位置指针 ← 结束位置指针 - 1;
```

【C++语言描述】

```
void sortFlag(char flagArray[], unsigned size) {
    int less = 0, more = size - 1, index = 0;
    while (index < more) {
        switch (flagArray[index]) {
            case 'R':    // `index` element is R -> swap with next `less` element.
                std::swap(flagArray[index++], flagArray[less++]);
                break;
            case 'W':    // `index` element is W -> pass.
                index++;
                break;
            case 'B':    // `index` element is B -> swap with previous `more` element.
                // BUT we do NOT know which type previous `more` element is,
                // so we need to make `index` still.
                std::swap(flagArray[index], flagArray[more--]);
                break;
        }
    }
}
```

5. 全部过桥最短时间问题。

【算法设计】

将问题抽象成给定 n 个人和他们各自通过桥所需的时间 T[n]，求最短耗时的问题。由于两个人一起走路的速度等于其中较慢那个人的速度，所以除了走的最快的人，其他的人在桥上耗费的时间都要算入过桥时间，加之每次要有一个人回去送手电筒，所以**让最快的人来回跑**将是最省时间的做法。算法需要找到速度最快的那个人（可以用第 1 题类似的方法），并且求出除了最快的人剩下人通过桥的总时间。当 n>2 时，最快的人需要往返 *n-2* 次桥，需要**在**（除了最快的人的）**总时间上加 *n-2* 倍的最快的人需要的时间即为答案**；n<2 时为时间最大的人的耗时。

【时间复杂度】

n 为人数，仅仅进行了一次循环来比较找出最小值并求和，故算法的时间复杂度为 O(*n*)。

【伪代码】

```
Algorithm: timeOfCrossingBridge
Input: 各自通过桥所需的时间 T[n], 人数 n
Output: 全部过桥的最短时间
    1.  if n < 2 then return T[n]的最大值;
    2.  index ← 0, sum ← 0;
    3.  i 循环 1 到 n
          3.1 if T[index] > T[i] then index ← i;
          3.2 sum ← sum + T[i];
    4.  sum ← sum + (n - 3) * T[index];   // n-3 是因为求和时多加了最快的人的时间
    5.  return sum;
```

【C++语言描述】

```
int timeOfCrossingBridge(const int timeOfEachPerson[], unsigned numberOfPeople) {
    if (numberOfPeople == 1) {    // Case - the number of people = 1
        return timeOfEachPerson[0];
    } else if (numberOfPeople == 2) {    // Case - the number of people = 2
        return std::max(timeOfEachPerson[0], timeOfEachPerson[1]);
    }
    // Case - the number of people >= 3
    int indexOfMinTimePerson = 0, sumTime = 0;
    for (int i = 0; i < numberOfPeople; ++i) {
        if (timeOfEachPerson[indexOfMinTimePerson] > timeOfEachPerson[i]) {
            // Find the fast person.
            indexOfMinTimePerson = i;
        }
        // Calculate the sum of the time of each person to cross the bridge.
        sumTime += timeOfEachPerson[i];
    }
    sumTime += (numberOfPeople - 3) * timeOfEachPerson[indexOfMinTimePerson];
    return sumTime;
}
```