第十二周实验报告

沈家成

1056 二哥吃糖

对象分析

本题对象较多，需要分析清楚之间的关系，才可以找出合适的数据结构表示。

主要对象为盒子和糖果，关系为糖果放在盒子里。如果以盒子为基本单位，将糖果储存在盒子中，就会遇到扩容，查找糖果的问题。

换个角度，以糖果为基本单位，储存糖果对应的盒子，那么找糖果相当方便，合并，吃糖的操作也简单了许多，最后就剩下了盒子内糖果数目的排序，可以利用查找树达到较好的时间复杂度。

散列表的使用

糖果与盒子的关系，用散列表可以达到很好的效果。

用 sweet[i] 表示第 i 个糖果在第几个盒子，用 box[i] 表示第 i 个盒子有多少个糖果。用 mergebox[i] 表示第 i 个糖果合并到了哪个盒子。

合并操作

1. 通过 mergebox 和 sweet 找到对应的盒子
2. 若两个糖果在同一个盒子，或者有一个糖果的盒子为空，直接结束
3. 合并两个盒子，调整糖果对应盒子

示例如下：

index	1	2	3	4	5
sweet	1	2	3	4	5
box	1	1	1	1	1
mergebox	0	0	0	0	0

```
C 1 2
```

index	1	2	3	4	5
sweet	1	2	3	4	5
box	2	0	1	1	1
mergebox	0	1	0	0	0

糖果 2 对应的盒子合并到了糖果 1 对应的盒子。

数量迁移到盒子 1

```
box[sweet[1]] += box[sweet[2]];
box[sweet[2]] = 0;
```

糖果二对应的盒子迁移到盒子 1

```
mergebox[sweet[2]] = sweet[1];
```

同样的，另一个合并操作。

```
C 3 4
```

index	1	2	3	4	5
sweet	1	2	3	4	5
box	2	0	2	0	1
mergebox	0	1	0	1	0

```
C 1 5
```

index	1	2	3	4	5
sweet	1	2	3	4	5
box	3	0	2	0	0
mergebox	0	1	0	1	1

```
C 2 5
```

index	1	2	3	4	5
sweet	1	1	3	4	1
box	3	0	2	0	0
mergebox	0	1	0	1	1

由于糖果 2 和糖果 5 都合并到了盒子 1 ，因此 mergebox 不为 0 ，这就需要先找到对应的盒子。

```
while (mergebox[sweet[2]] != 0) {
    sweet[2] = mergebox[sweet[2]];
}
while (mergebox[sweet[5]] != 0) {
    sweet[5] = mergebox[sweet[5]];
}
```

由于大多是$O(1)$的赋值操作，合并盒子的操作时间复杂度较低。

吃糖操作

1. 通过 mergebox 找到糖果对应的盒子
2. 把这个盒子的数量置零

示例如下：

```
D 5
```

index	1	2	3	4	5
sweet	1	1	3	4	1

index	1	2	3	4	5
box	0	0	2	0	0
mergebox	0	1	0	1	1

```
while (mergebox[sweet[5]] != 0) {
    sweet[5] = mergebox[sweet[5]];
} //sweet[5] = 1;
box[sweet[5]] = 0; //box[1] = 0;
```

大多数操作是赋值，因此时间复杂度较好。

查找树

寻找操作

寻找操作要求返回第 i 大的盒子的糖果数，也就是需要对 box 数组进行排序，需要一个插入、删除、查找操作都足够优秀的数据结构，因此选择二叉查找树。

二叉树的平衡以高度为标准，为了适应此题，可以以结点数代替高度，方便查找第 i 大的结点。

```
t->height = max(getHeight(t->left), getHeight(t->right)) + 1;
```

改成

```
t->height = getHeight(t->left) + getHeight(t->right) + 1;
```

与合并、吃糖操作融合

合并操作中，加入

```
tree.remove(box[sweet[num1]]);
tree.remove(box[sweet[num2]]);
box[sweet[num1]] += box[sweet[num2]];
tree.insert(box[sweet[num1]]);
```

吃糖操作中，加入

```
tree.remove(box[sweet[num1]]);
```

这样，就可以在合并、吃糖操作中维护这个二叉查找树，输出结果时直接查找第 i 个即可。

1228 Matrix Sum

预备定理

> 奇数 + 奇数 = 偶数
> 奇数 + 偶数 = 奇数
> 偶数 + 偶数 = 偶数

如果用 1 表示奇数， 0 表示偶数，就可以表达为：

> 1 + 1 = 0
> 1 + 0 = 1
> 0 + 0 = 0

这就是不进位的二进制加法。

因此，对于输入的数据，只需要保留它的奇偶信息。

如：

$$\begin{matrix} 802 & 516 & 936 \\ 906 & 150 & 155 \\ 39 & 221 & 557 \end{matrix}$$

可化为：

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{matrix}$$

子矩阵和

单行情况

先从简单的开始，如果只有一行，如何求哪些子矩阵和为奇数呢？

$$0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$$

通过试验，可以发现 0 对于和是否奇数并没有影响，1 才是决定奇偶的因素，0 决定可以和 1 配对成多少个子矩阵。

对于第一个 1 ，前面有三个 0 ，后面有一个 0 ，根据排列组合，前面可以不取、取第三个、取第三第二个、取第三第二第一个，共四种情况，后面可以不取、取第一个，共两种情况。由乘法原理，共有$4 \times 2 = 8$种情况

对于多个 1 ，只需要将奇数个 1 之和，等价成 1 即可。如对于$1\ 0\ 1\ 1$ ，前面有三个 0 ，后面有一个 0 ，所以总共有$4 \times 2 = 8$种情况。

多行

单行情况分析清楚了，相当于连续列分析完成了，剩下的就是连续行。

$$\begin{matrix} 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0 \\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0 \end{matrix}$$

这两行共同的子矩阵怎么判断呢？回想起之前的预备定理，将两行加起来，正好筛选出了奇数个奇数。

$$0\ 1\ 0\ 0\ 1\ 1\ 1\ 0$$

接下来，用单行的方法分析即可。

代码流程

1. 计算当前行的奇数子矩阵和个数
2. 将下一行的矩阵加到当前行，计算奇数子矩阵个数
3. 继续 2 ，直到最后一行
4. 将当前行往下移一行，继续 1 ，直到最后一行。

偶数和

对于$n \times n$的矩阵，子矩阵总数固定，为$\frac{n^2(n+1)^2}{4}$，用奇数个数减去就可以得到偶数个数。

1634 Sort, sort and sort

主要是要理清排序的时候哪里发生了比较，再计数即可。

堆排序

建堆

```
1 3
1 3 2
1 3 2 4
1 3 2 4 5
```

4 次比较

出队

```
5 3 2 4
2 3 5 4
```

第一次出队，2次比较

```
4 3 5
3 4 5
```

第二次出队，2次比较

```
5 4
```

第三次出队，1次比较

```
4
```

第四次出队，0次比较

共 9 次比较。

归并排序

第一次分割

```
1 3 | 2 4 5
```

第二次分割

```
1 || 3 | 2 || 4 5
```

第三次分割

```
1 || 3 | 2 || 4 ||| 5
```

第一次归并

```
1 || 3 | 2 || 4 5
```

1 次比较

第二次归并

```
1 3 | 2 4 5
```

2 次比较

第三次归并

```
1
1 2
1 2 3
1 2 3 4 5
```

3 次比较

共 6 次比较

快速排序

寻找第一个元素位置

```
 1   3   2   4   5
low             high
 1   3   2   4   5
low         high
 1   3   2   4   5
low     high
 1   3   2   4   5
low high
 1 | 3   2   4   5
```

4 次比较

```
 1 | 3   2   4   5
    low         high
 1 | 3   2   4   5
    low     high
 1 | 3   2   4   5
    low high
 1 | 2 | 3 | 4   5
    low high
```

3 次比较

```
 1 | 2 | 3 | 4   5
            low high
```

1 次比较

总共 8 次比较

代码具体实现

计数的操作要尽量和比较操作靠近

原本为:

```
for(;hole > 1 && x < array[hole / 2]; hole /= 2)
{
    array[hole] = array[hole / 2];
}
```

改为:

```
for(;hole > 1; hole /= 2)
{
    ++cmp_ctr;
```

```
        if(x < array[hole / 2]) {
            array[hole] = array[hole / 2];
        } else {
            break;
        }
    }
```

将条件判断分拆出来，插入比较计数。