第十四周实验报告

沈家成

1593 Mouse

基本思路

每次是排名相邻的老鼠进行较力，因此排序肯定是需要的。

刚开始是乱序，因此选用快速排序是最好的。但是如果之后每次都重新快排，肯定会超时，要充分利用已经排序过的信息。

胜者组成的新序列，和败者组成的新序列，由于都是加上同样的分数，因此依然是有序的，可以用归并排序，达到很高的效率。

平局比较麻烦。有两种思路：

1. 平局者也组成一个有序的新序列，然后三个序列归并
2. 按一定规则将平局者放入胜者序列和败者序列，保证有序性即可

最后选择了后者，原因如下：

- 只分为胜者和败者序列的话，序列长度固定为N，便于储存。如果平局者单独存放一个数组，长度不固定，最大可能到2N
- 三数列归并排序，如果直接分析各种情况，情况复杂，容易有欠考虑的情况
- 如果先归并两个序列，再归并两个序列，临时数组的开销和赋值操作的开销较大

比较运算符重载

快速排序和归并排序已经有成熟的代码，不赘述。需要注意的是结构体比较运算符的重载。

为了方便，将老鼠的得分、力量值、编号组成一个结构体，是理所当然的。为了直接应用快速排序和归并排序，需要重载比较运算符。题目中提到，得分相同时，编号小的考前，因此比较运算符这样重载：

```
struct Mouse
{
    unsigned long score, strength, id;
    bool operator > (Mouse & m2) {
        if (score == m2.score)
            return id < m2.id;
        return this->score > m2.score;
    }
}
```

得分相同时，比较编号。得分不同时，依然按得分排序。

平局情况分析

考虑如下特殊情况：

score:	**40**	**40**	**40**	**40**	**40**	**40**
strength	0	0	0	0	0	1

那么一轮过后，分数更新成：

score:	**41**	**41**	**41**	**41**	**40**	**42**

score:	41	41	41	41	40	42
strength	0	0	0	0	0	1

如果直接将胜者归入胜者序列，就会发生大数在后面的情况：

winner:	41	41	42

败者序列也是类似情况。主要原因是在原本的相对顺序上加上了不同的得分，可能破坏这种顺序。

因此，每次归入序列时，要先进行检查：

```
while (k > 0 && winner[j] > winner[j-1]){
    tmp = winner[j];
    unsigned long p = j;
    while(p > 0 && tmp > winner[p-1]) {
        winner[p] = winner[p-1];
        --p;
    }
    winner[p] = tmp;
}
```

由于这样的情况毕竟较少，直接插入到相应位置即可。

1635 经济出行计划

问题归类

这是一个单源最短路径问题，选用经典的 Dijkstra 算法就可以得到正确答案，问题主要在优化时间复杂度。

性能瓶颈

最耗时的地方，在于寻找 V - S 集合中的最小距离。直接顺序查找，单次时间复杂度为$O(|V|)$，造成总的时间复杂度为$O(|V^2|)$。

其实，每次更新的距离是有限的，所以直接顺序查找进行了重复工作，因此选用二叉查找树来进行最小距离的查找。

二叉查找树

二叉查找树的 insert 和 remove 操作可以直接使用，只需要再添加一个查找最小值的函数即可。

```
Type minimum() {
    AANode * p = root;
    if (p == NULL)
        return Type(0,0);
    while(p->left != NULL)
        p = p->left;
    return p->data;
}
```

根据二叉查找树的定义，一直往左下角找，就可以找到最小值了。

改进后的 Dijkstra 算法

二叉查找树找最小值的时间复杂度为$O(log|V|)$，因此总的时间复杂度就优化到了$O(|V|log|V|)$。

```
void search(TypeOfEdge noEdge, TypeOfVer end) const {
    TypeOfEdge * distance = new TypeOfEdge[Vers];
    bool * known = new bool [Vers];
```

```
    long u, i ,j;
    edgeNode * p;
    TypeOfEdge min;
    costNode min_node(0, 0);

    AATree< costNode > tree;

    for (i = 0; i < Vers; ++i) {
        known[i] = false;
        distance[i] = noEdge;
    }
    distance[0] = 0;

    for (i = 0; i< Vers; ++i) {
        min_node = tree.minimum();
        tree.remove(min_node);
        u = min_node.ver;
        min = min_node.cost;
        if (u == end)
            break;
        known[u] = true;
        for (p = verList[u].head; p != NULL; p = p->next) {
            if (known[p->end])
                continue;
            if (distance[p->end] == noEdge) {
                distance[p->end] = min + p->weight;
                tree.insert(costNode(distance[p->end], p->end));
                continue;
            }
            if (distance[p->end] > min + p->weight) {
                tree.remove(costNode(distance[p->end], p->end));
                distance[p->end] = min + p->weight;
                tree.insert(costNode(distance[p->end], p->end));
            }
        }
    }
    cout << distance[end] << endl;
}
```

主要更改在两个地方：

1. 最小值通过二叉查找树获取，获取后要记得删除最小值
2. 更新最小距离时，先在二叉查找树中删去旧的最小距离，再插入新的最小距离

结构体比较运算符重载

为了在二叉树中方便地储存距离信息，自然地想到通过结构体将距离和结点编号绑定在一起。通过结构体的比较运算符重载，可以直接通过模板特化使用二叉树。

```
struct costNode {
    TypeOfEdge cost;
    long ver;
    bool operator < (costNode & c2) {
        if (cost == c2.cost)
            return ver < c2.ver;
        return cost < c2.cost;
    }
}
```

由于二叉树不允许储存相同的数据，因此可以通过编号来区分最小距离相等的结点。

1999 二哥找宝箱

暴力枚举

题目里说最多有 5 个宝箱，因此如果知道了宝箱之间的最短距离，以及起点到每个宝箱的最短距离，就可以直接暴力枚举，尝试每种选择，从而得出最小步数。

bfs

迷宫两个点之间的最小距离，可以通过广度优先搜索得到。

```
int search(Point & begin, Point & end) {
    MyQueue<Point> queue;
    for (int i = 1; i <= N; ++i) {
        for (int j = 1; j <= M; ++j) {
            visited[i][j] = false;
        }
    }
    begin.step = 0;
    queue.enQueue(begin);
    while(!queue.isEmpty()) {
        Point now = queue.deQueue();
        visited[now.x][now.y] = true;
        Point next;
        for (int i = 0; i < 4; ++i) {
            next.x = now.x + dx[i];
            next.y = now.y + dy[i];
            if (next.x >= 1
                    && next.x <= N
                    && next.y >= 1
                    && next.y <= M) {
                if (visited[next.x][next.y])
                    continue;
                next.type = map[next.x][next.y];
                visited[next.x][next.y] = true;
                if (next.type != -1) {
                    next.step = now.step + 1;
                    if (next.x == end.x && next.y == end.y)
                        return next.step;
                    queue.enQueue(next);
                }
            }
        }
    }
    return -1;
}
```

使用队列储存要搜索的结点，找到终点就输出步数。

全排列数的生成

5 个宝箱，全排列有$5 \times 4 \times 3 \times 2 \times 1 = 120$种，最好能够生成一个这样的序列，方便枚举。

```
bool end = false;
while (!end) {
    int distance = dp[5][path[0]];
    for (int i = 1; i < treasure_ctr; ++i) {
        distance += dp[path[i]][path[i-1]];
    }
    closest = closest < distance ? closest : distance;
    int i,j;
    for (i = treasure_ctr-2; i >= 0; --i) {
        if (path[i] < path[i+1])
            break;
        else if (i == 0)
            end = true;
    }

    for (j = treasure_ctr-1; j > i; --j) {
        if(path[j] > path[i])
            break;
```

```
    }

    swap(path,i,j);
    reverse(path,i+1,treasure_ctr-1);
}
```

生成全排列数的方法主要在于：

- 从后往前找第一对升序相邻元素$a_i < a_{i+1}$，找不到说明全排列完了，可以退出
- 从后往前，找第一个比a_i大的元素a_j，交换a_i, a_j
- 排列逆序倒置，就得到了下一个排列

生成的全排列数是这样的：

ith	1	2	3
1	0	1	2
2	0	2	1
3	1	0	2
4	1	2	0
5	2	0	1
6	2	1	0

这样，通过暴力枚举就可以得到最短路径了。