CH7 课后习题

CH7 课后习题

- 7.3
- 7.12
- 7.16
- 7.20
- 7.30
- 7.38
- 7.52

7.3

answer

1. 考虑 `a[i]` 与 `a[i+k]` 两数：由于两数最初在数列中为无序，因此两数之间将构成一个逆序，交换二者必将**至少**使数列减少由这二者所构成的1个逆序

2. 考虑 `a[j]`，（其中: `j < i` or `j > i+k`）：易知 `a[i]` 与 `a[i+k]` 两数之间的交换并不会改变它们与 `a[j]` 之间的前后关系，故这一交换**不会**令两数与 `a[j]` 产生或消除逆序

3. 考虑 `a[j]`，（其中：`i < j < i+k`）：由于**大于**与**小于**为**严格全序关系**，则跟据严格全序关系中的**传递性**并结合反证法可推出：（1）若交换前 `a[j]` 与 `a[i+k]` 之间为顺序，则 `a[j]` 与 `a[i]` 之间必为逆序，否则，`a[i]` 与 `a[i+k]` 之间初始应为顺序，与题设矛盾；（2）同理，若交换前 `a[j]` 与 `a[i]` 之间为顺序，则 `a[j]` 与 `a[i+k]` 之间必为逆序；综上不难看出，交换 `a[i]` 与 `a[i+k]` **只能**使两数与两数之间的数所构成的逆序总数**减少或保持不变**

4. 综上：（1）若 `a[j]` 同时与两数构成逆序时，此次交换能消除三数之间构成的逆序，即对单数讨论最大消除2个逆序，而 `a[i]` 与 `a[i+k]` 之间有 `k-1` 个数，故能消除逆序**最大**为 `2(k-1) + 1 = 2k-1` 个；（2）此外的情况，交换 `a[i]` 与 `a[i+k]` 两数都不会使它们与 `a[j]` 直接的逆序总数发生改变，故交换 `a[i]` 与 `a[i+k]` 两数**最少**能使逆序总数减少 `1` 个

7.12

answer

一个标准的堆排序分为 `buildHeap` 与 `deleteMax(|Min)` 两个过程，对于一个预排序后的数列，第一阶段 `buildHeap` 的时间复杂度为 `O(n)`；第二阶段在每一次删除根元素后，作为堆末尾的数（同时也为数列最大（小）数）都将被交换到堆顶然后执行**下滤**操作，且每一次下滤操作都必将下滤到叶子节点，由此有：

对于 $2^{k-1} \leq n < 2^k$，其中 $k \in \mathbb{N}^*$，有：

$$k = \lceil \log_2{(n+1)} \rceil$$

第二阶段的执行时间：

$$\sum_{i=1}^{n} 2k = \sum_{i=1}^{n} 2\lceil \log_2{(i+1)} \rceil < 2n\log_2 n$$

故对预排序数列堆排序的时间复杂度为 $O(n\log n)$

7.16

analysis

通过结合使用**栈**与**队列**，我们可以实现对归并排序的解递归，见 `code1` ；当然，若不纠结非递归的分割方式是否与递归的分割方式相同，我们可以用一种更粗糙同时也更简单的方式实现非递归的归并排序，见 `code2` ；

merge function

```
template <typename Comparable>
void merge( vector<Comparable> & v, vector<Comparable> & tmpArray, \
            int leftPos, int rightPos, int rightEnd ) {

    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while (leftPos <= leftEnd && rightPos <= rightEnd) {
        if (v[leftPos] < v[rightPos]) {
            tmpArray[tmpPos++] = v[leftPos++];
        } else {
            tmpArray[tmpPos++] = v[rightPos++];
        }
    }

    while (leftPos <= leftEnd) {
        tmpArray[tmpPos++] = v[leftPos++];
    }

    while (rightPos <= rightEnd) {
        tmpArray[tmpPos++] = v[rightPos++];
    }

    for (int i = 0; i < numElements; ++i, --rightEnd) {
        v[rightEnd] = tmpArray[rightEnd];
    }
}
```

code1

```
template <typename Comparable>
void mergeSort(vector<Comparable> & v) {

    vector<Comparable> tmpArray(v.size());

    queue<int> start_q, end_q;
    stack<int> start_s, end_s;
    int start = 0, end = v.size() - 1;
    start_q.push(start);
    end_q.push(end);

    while (!start_q.empty() && !end_q.empty()) {
        start = start_q.front();
        end = end_q.front();
        start_q.pop();
        end_q.pop();
        start_s.push(start);
        end_s.push(end);
        if (start < end) {
            int mid = (start + end) / 2;
            start_q.push(start);
            end_q.push(mid);
            start_q.push(mid + 1);
            end_q.push(end);
        }
    }

    while (!start_s.empty() && !end_s.empty()) {
        start = start_s.top();
        end = end_s.top();
        start_s.pop();
        end_s.pop();
        if (start < end) {
            int mid = (start + end) / 2;
            merge(v, tmpArray, start, mid + 1, end);
        }
    }
}
```

code2

```
template <typename Comparable>
void mergeSort(vector<Comparable> & v) {
    vector<Comparable> tmpArray(v.size());
    int n = v.size();

    for (int splitSize = 1; splitSize < n; splitSize *= 2) {
        for (int i = 0; i < n; i += splitSize*2) {
            int leftPos = i;
            int rightPos = i + splitSize;
            int rightEnd = min(i + splitSize*2 - 1, n - 1);
            merge(v, tmpArray, leftPos, rightPos, rightEnd);
        }
    }

}
```

7.20

code

按书中实现方法，使用**三数中值分割**的策略，实现代码如下：

```
template <typename Comparable>
Comparable median3(vector<Comparable> & a, int left, int right) {
    int center = (left + right) / 2;
    if (a[center] < a[left])
        swap(a[left], a[center]);
    if (a[right] < a[left])
        swap(a[left], a[right]);
    if (a[right] < a[center])
        swap(a[center], a[right]);

    swap(a[center], a[right - 1]);
    return a[right - 1];
}

template <typename Comparable>
void quickSort(vector<Comparable> & a, int left, int right) {
    if (left >= right)
        return;
    Comparable pivot = median3(a, left, right);
    int i = left, j = right - 1;
    for ( ; ; ) {
        while (a[++i] < pivot) {}
        while (pivot < a[--j]) {}
        if (i < j)
            swap(a[i], a[j]);
        else
            break;
    }

    swap(a[i], a[right - 1]);

    quickSort(a, left, i - 1);
    quickSort(a, i + 1, right);
}

template <typename Comparable>
void quickSort(vector<Comparable> & a) {
    quickSort(a, 0, a.size() - 1);
}
```

a.answer

对于一个预排序后的数列，每次选取的枢纽元恰好位于数列的中间。不妨假设两个子数组恰好各为原数组的一半大小（通常这会导致比原结果较高的估计，但我们只关心大 O 的答案，因此结果仍在允许的范围内）：

$$T(N) = 2T(N/2) + cN$$

令等式两边同时除以 N：

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c$$

反复套用该公式：

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c = \frac{T(N/4)}{N/4} + c + c = \cdots = \frac{T(1)}{1} + c\log N$$

由此得到：

$$T(N) = cN\log N + N = \Theta(NlogN)$$

故对于一个预排序后的数列，快速排序的时间复杂度为 $O(N\log N)$

b.answer

与 a 同理，对于一个反排序后的数列，快速排序的时间复杂度同样为 $O(N\log N)$

当然，枢纽元的选择将会对这两种情况的分析产生较大的影响。若将枢纽元选为数列的第一个元素或者最后一个元素，那么对于一个预排序或反排序的数列，快速排序的时间复杂度都将为 $O(N^2)$。

c.answer

对于一个长度为 N 随机的数列，我们不妨假设数列中的每个元素的大小都是等可能的，因此它们均有 $\frac{1}{N}$ 的概率。

从而通过考虑每个可能的子集规模的时间开销并对它们求平均值，我们可以得到分割后两个子集的执行时间平均值为：

$$\frac{1}{N}\sum_{j=0}^{N-1}T(j)$$

从而：

$$T(N)=\frac{2}{N}[\sum_{j=0}^{N-1}T(j)]+cN$$

等式两边乘 N：

$$NT(N)=2[\sum_{j=0}^{N-1}T(j)]+cN^2 \quad ①$$

当 $N>1$ 时，有：

$$(N-1)T(N-1)=2[\sum_{j=0}^{N-2}T(j)]+c(N-1)^2 \quad ②$$

令 ① 减去 ②：

$$NT(N)-(N-1)T(N-1)=2T(N-1)+2cN-c$$

舍去常数项并整理得：

$$NT(N)=(N+1)T(N-1)+2cN$$

即：

$$\frac{T(N)}{N+1}=\frac{T(N-1)}{N}+\frac{2c}{N+1}$$

反复套用该公式，叠缩求和得：

$$\frac{T(N)}{N+1}=\frac{T(N-1)}{N}+\frac{2c}{N+1}=\frac{T(N-2)}{N-1}+\frac{2c}{N}+\frac{2c}{N+1}=\cdots=\frac{T(1)}{2}+\sum_{i=3}^{N+1}\frac{2c}{i}$$

该和大约为：

$$log_e(N+1)+\gamma-3/2$$

其中 γ 为欧拉常数，故：

$$T(N) = O(NlogN)$$

7.30

求解下列递推关系：

$$T(N) = (1/N)[\sum_{i=0}^{N-1} T(i)] + cN, \quad T(0) = 0$$

answer

有：

$$NT(N) = [\sum_{i=0}^{N-1} T(i)] + cN^2 \quad ①$$

当$N > 1$时，有：

$$(N-1)T(N-1) = [\sum_{i=0}^{N-2} T(i)] + c(N-1)^2 \quad ②$$

令 ① 减去 ②：

$$NT(N) - (N-1)T(N-1) = T(N-1) + cN^2 - c(N-1)^2$$

整理得：

$$NT(N) = NT(N-1) + 2cN - c$$

即：

$$T(N) = T(N-1) + 2c - \frac{c}{N}$$

进行叠缩：

$$T(N-1) = T(N-2) + 2c - \frac{c}{N-1}$$

$$\dots$$

$$T(1) = T(0) + 2c - \frac{c}{1} = c$$

将上述各式相加：

$$T(N) = T(0) + 2cN - c\sum_{i=1}^{N}\frac{1}{i} < 2cN$$

故:

$$T(N) = O(N)$$

值得一提的是，该递推证明的结果即为书中 `quickSelect` 的平均运行时间。

7.38

analisys

对于平面上的 N 个不同点，其中每两个不同的点便可确定一条直线，我们可以得到 $C_N^2 = \frac{N(N-1)}{2}$ 种组合。

首先定义出 `Point` 类与 `Line` 类，为了方便判断两直线是否相同，我们在 `Line` 类中同时定义直线的斜率与截距；同时，为了能够让我们更方便的判断两个 `Point` 对象是否相同，我们在 `Point` 类中重载了 `==` 运算符；为了让 `Line` 对象能够参与到比较中，我们需要在 `Line` 类中重载 `<` 运算符；

class Point

```
struct Point {
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) { }

    bool operator == (const Point & rhs) const {
        return x == rhs.x && y == rhs.y;
    }
};
```

class Line

```
class Line {
private:
    Point p1, p2;
    double slope;
    double intercept; // when the line is not vertical, use y intercept;
                          // otherwise, use x intercept;
public:
    Line(const Point & p1, const Point & p2);

    double computeSlope( ) const;

    double operator < (const Line & rhs) const;
    double getSlope( ) const { return slope; }
    double getIntercept( ) const { return intercept; }
    Point getP1( ) const { return p1; }
    Point getP2( ) const { return P2; }
};
```

构造函数：

```
Line::Line(const Point & p1, const Point & p2) : p1(p1), p2(p2) {
    slope = computeSlope();
    if (slope == std::numeric_limits<double>::infinity())
        intercept = p1.x;
    else
        intercept = p1.y - slope * p1.x;
}
```

`computeSlope` 函数用于计算直线的斜率，当直线为垂直时，我们将斜率定义为 `std::numeric_limits<double>::infinity()`，实现如下：

```
double Line::computeSlope( ) const {
    if (p1.x == p2.x)
        return std::numeric_limits<double>::infinity();
    else
        return (p2.y - p1.y) / (p2.x - p1.x);
}
```

排序的目的主要是要把相同的直线聚集到一起，因此我们可以如此设计 `Line` 类中 `<` 的逻辑：

- 若两直线的斜率不相同，则直接比较斜率大小；
- 若两直线的斜率相同，则比较截距大小；

实现如下：

```
double Line::operator < (const Line & rhs) const {
    if (slope != rhs.slope)
        return slope < rhs.slope;
    else
        return intercept < rhs.intercept;
}
```

接下来我们考虑如何对 `Line` 进行排序。

已知直线数量的量级为 $O(N^2)$，要使算法的时间复杂度达到 $O(N^2logN)$，我们需要使用一种对 n 个输入平均时间复杂度为 $O(nlogn)$ 的算法（例如：快速排序等，此处具体选择何种排序算法并非题目的重点，因此此处不写出具体的排序步骤，而使用 STL库中的 `sort` 函数进行替代，其底层实现基于快速排序）。实现如下：

```cpp
int main() {
    vector<Point> points;
    vector<Line> lines;
    vector<Point> collinear;

    // first input the coordinates of points
    int numPoints; cin >> numPoints;
    points.reserve(numPoints);
    for (int i = 0; i < numPoints; ++i)
        cin >> points[i].x >> points[i].y;

    // generate all possible lines
    for (int i = 0; i < numPoins; ++i)
        for (int j = i + 1; j < numPoints; ++j)
            lines.push_back(Line(points[i], points[j]));

    // sort the lines
    sort(lines.begin(), lines.end());

    // find the collinear points
    for (size_t i = 0; i < lines.size(); ++i) {
        if (i == 0 || \
            lines[i].getSlope != lines[i-1].getSlope() || \
            lines[i].getIntercept() != lines[i-1].getIntercept()) {

            if (collinear.size() > 3) {
                printPoints(collinear);
                cout << endl;
            }
            collinear.clear();
        }

        collinear.push_back(lines[i].getP1());
        collinear.push_back(lines[i].getP2());
    }

    return 0;
}
```

排序后依次找出的共线的点将会被存储在 `collinear` 中，最后我们写出用于输出各共线点的函数 `printPoints` ：

```
void printPoints(vector<Point> collinear) {
    vector<Point>::iterator it;
    int count = 0;
    it = unique(collinear.begin(), collinear.end());
    for (auto itr = collinear.begin(); itr != it; ++itr)
        count++;
    if (count > 3)
        for (auto itr = collinear.begin(); itr != it; ++itr)
            cout << "(" << itr->x << ", " << itr->y << ") ";
}
```

以上还要注意的是，`collinear` 中所存储的点有部分可能会出现重复，因此我们可以通过使用 `unique` 函数对 `vecotr` 容器进行去重。

至此，代码结束。

7.52

answer

引理1（书中已给出证明）：**具有 L 片树叶的二叉树的深度至少为** $\lceil \log L \rceil$

引理2：**任何具有 L 片树叶的任意二叉树的平均深度至少为** $\log L$

证明：

由 引理1 马上就可推出该结论。

当然，我们也可以通过**强归纳法**证明：

- 当 $L = 1$ 时，显然成立
- 假设该结论对于所有树叶数少于 L-1 的树都成立：

考虑一颗具有 L 片树叶且平均叶节点深度最小的树，并假设其左子树有 L_L 片树叶，右子树有 L_R 片树叶，由于要使平均节点深度最小，易知当 L 大于 1 时根节点的左右子树都不应为空，则：

其左子树叶节点深度总和为：

$$L_L(1 + \log L_L)$$

同理，右子树叶节点深度总和为：

$$L_R(1 + \log L_R)$$

则该树的叶节点深度总和为：

$$L_L(1 + \log L_L) + L_R(1 + \log L_R) = L + L_L \log L_L + L_R \log L_R$$

因 $xlogx$ 是个凸函数，我们有：

$$f(x) + f(y) \geq 2f(\frac{x+y}{2})$$

故：

$$L + L_L \log L_L + L_R \log L_R \geq L + 2 * \frac{L_L + L_R}{2} \log\left(\frac{L_L + L_R}{2}\right) = L + L \log \frac{L}{2} \geq L \log L$$

故平均叶节点深度至少为 $\log L$，证毕。

定理1：**只使用元素间比较的任何排序算法平均至少需要$\lceil \log(N!) \rceil$次比较**

证明：

对 *N* 个元素排序的决策树必有 *N!* 片树叶。由 引理2 可知，该决策树的平均深度至少为 $\log N!$
，故平均比较次数至少为 $\lceil \log N! \rceil$。

定理2：**任何基于比较的排序算法平均都需要 $\Omega(N \log N)$ 次比较**

证明：

由 定理1 可知，需要 $\lceil \log N! \rceil$ 次比较，而：

$$\log(N!) = \log\left(N(N-1)(N-2)...(2)(1)\right)$$

$$= \log N + \log(N-1) + \log(N-2) + ... + \log 2 + \log 1$$

$$\geq \log N + \log(N-1) + \log(N-2) + ... + \log(N/2)$$

$$\geq \frac{N}{2} \log \frac{N}{2}$$

$$= \Omega(NlogN)$$

故平均比较次数至少为 $\Omega(N \log N)$。