

# 排序算法与分治1

排序是最基础的算法之一。本文介绍快速排序、归并排序和它们的应用。同时，排序算法也是分治和递归思想的一个绝佳例子，有利于我们对分治思想有个初步了解。本文的排序算法只针对数组上的32位整数进行排序。

## 基本概念理解

### 递归

递归的定义非常简单，只要出现一个函数自己调用自己，这就是递归。当一个C++程序运行起来的时候，在操作系统的视角里，这是在启动一个“进程”（process）。进程里有一片区域叫做“栈区域”，栈区域是用来存储函数调用相关的信息。

### 栈的极简介绍

栈（Stack）是一种后进先出（LIFO）的线性数据结构，元素只能从栈顶插入和删除。基本操作包括：

- **push**：入栈
- **pop**：出栈
- **top**：查看栈顶元素
- **empty**：判断是否为空

栈常用于函数调用、括号匹配、浏览器历史记录等。

### 递归的操作系统实现

在操作系统层面，递归的实现主要依赖于**函数调用栈**（call stack）来管理函数调用的过程。每次递归调用函数时，操作系统通过函数调用栈保存当前函数的状态，包括局部变量、函数参数、返回地址等。

以下是递归在操作系统层面的具体步骤：

- 栈帧（Stack Frame）创建**: 当一个函数被调用时，操作系统为该函数创建一个**栈帧**，栈帧是函数调用栈中的一个区域，用于存储该函数的局部变量、函数参数、返回地址等信息。
- 递归调用时的栈帧管理**: 当递归调用发生时，操作系统会在函数调用栈上**压入（push）**一个新的栈帧，保存当前函数的上下文信息（包括当前递归函数的局部变量和参数）。这个过程会不断重复，直到满足递归终止条件。
- 函数返回时的栈帧弹出**: 当递归到达终止条件时，递归函数开始返回。这时，操作系统会将当前的栈帧**弹出（pop）**函数调用栈，恢复之前的函数上下文（包括返回地址、局部变量等）。函数返回会逐级向上传递，直到所有递归层次都返回。
- 递归深度与栈空间限制**: 由于每次递归调用都会创建一个新的栈帧，栈的大小是有限的。如果递归过深，栈帧的数量可能超过操作系统为栈分配的空间，导致**栈溢出（stack overflow）**。这是递归在操作系统层面的一个潜在问题，尤其在递归层次非常深的情况下。

栈帧的具体内容:

- **返回地址**: 当递归调用结束时, 程序需要知道返回到上一级调用的位置 (即递归调用点)。
- **函数参数**: 每次递归调用会将当前的参数保存在栈中, 确保不同层次的调用可以独立维护各自的参数。
- **局部变量**: 递归中的局部变量在每次递归调用中会保存在当前的栈帧中, 保证每一层递归都能独立使用这些变量。

## 分治

分治一般是指, 将某个大问题划分为若干个小问题, 假设问题小到一定程度, 就可以不需要继续划分而直接解决, 那么划分有限次一定能停下来, 接着通过汇总小问题的结果, 反过来求解大问题。快速排序和归并排序都用到了分治的思想, 都是将大问题划分为小问题, 然后递归解决小问题, 接着就解决了原本的大问题。

## 快速排序

快速排序的思想是, 先随机选择数组中的一个数 $x$ , 然后通过比较和交换, 使得整个数组小于 $x$ 的数全部放在左边, 大于 $x$ 的数全部放在右边。左右两边各自再递归地进行排序。由于小于和大于 $x$ 的部分已经完全分开, 而且规模一定小于整体规模, 所以很自然的就想到可以递归地去解决。

例题[https://blog.csdn.net/qg\\_46105170/article/details/113790305](https://blog.csdn.net/qg_46105170/article/details/113790305)

```
1  #include <iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int a[N];
6
7  void quick_sort(int l, int r) {
8      if (l >= r) return;
9
10     int piv = a[l + (r - l >> 1)];
11     int i = l, j = r;
12     while (i <= j) {
13         while (a[i] < piv) i++;
14         while (a[j] > piv) j--;
15         if (i <= j) swap(a[i++], a[j--]);
16     }
17
18     quick_sort(l, j), quick_sort(i, r);
19 }
20
21 int main() {
22     int n;
23     scanf("%d", &n);
24     for (int i = 0; i < n; i++) scanf("%d", &a[i]);
```

```

25
26     quick_sort(0, n - 1);
27     for (int i = 0; i < n; i++) printf("%d ", a[i]);
28 }

```

注意在这个模板里，划分步骤结束之后，整个数组会分为 $[l, j]$ ,  $I$ ,  $[i, r]$ 这几个部分，其中 $I$ 这个中间部分有可能存在也有可能不存在，要看最后一步 $i, j$ 的情况。但是如果 $I$ 非空，那么其一定只含1个数。

类似于快速排序这种算法，最实用的学习方法就是找一个高效的模板进行背诵。上面这篇模板是可以信赖的。尤其要注意各个 $<$ ,  $\leq$ 等符号不要写错。为了证明它的正确性，我们可以尝试证明以下几点：

1. while循环一定会退出。这是最显然的，因为每一轮循环两个指针都一定会移动至少1步
2. 循环退出的时候， $a[l : j] \leq piv, a[i : r] \geq piv$ 。容易看出，每轮循环在 `if` 语句之前， $i$ 会停留在大于等于 $piv$ 的数之上，而 $j$ 会停留在小于等于 $piv$ 的数之上，所以如果循环退出的时候，存在某个 $k$ 使得 $a[k] > piv, k \leq j$ ，而 $k$ 必然是 $i$ 曾经扫到过的地方，它一定会被换成一个小于等于 $piv$ 的数，这样就矛盾了。
3. 最后就是要证明递归求解的时候，一定是在求解一个规模更小的问题。这也是显然。

由上面三点可以知道算法正确。

如果学有余力，可以考虑代码的任何部分被修改，会不会出现死循环、爆栈、答案错误等问题，并且思考上面的3点哪些会被破坏。

快速排序的期望（平均）时间复杂度是 $O(n \log n)$ ，最坏时间复杂度是 $O(n^2)$ ，如果选取pivot每次都选到了全局最小或者最大，导致划分不够平均，那么有可能会达到最坏时间复杂度。期望空间复杂度为 $O(\log n)$ ，最坏空间复杂度是 $O(n)$ ，这些额外空间消耗来自于递归栈。时间空间复杂度就不证明了。但这是个很好的时机介绍各个不同复杂度的含义：

1. 期望（平均）时间复杂度：设一个算法的输入的所有情况的集合是 $\Omega$ ，假设每个输入的概率均等，那么这个算法的时间复杂度的期望值，就是平均时间复杂度。
2. 最坏时间复杂度：如果不特殊说明，所有算法说的“时间复杂度”都是指的是最坏复杂度。最坏时间复杂度指的就是，对于能让算法达到时间最坏情况的输入的那个时间复杂度。
3. 均摊时间复杂度：指的是，某次操作总共进行 $n$ 次的总的时间复杂度再除以 $n$ 。因为有的操作可能会出现，单次最坏的情况下很慢，但如果操作多次，平均看来，每次却是很快的。针对这种情况，均摊复杂度更能反映实际情况。

## 快速选择

考虑这样的问题，给定一个长 $n$ 的序列 $a$ ，我们希望找到 $a$ 中从小到大排序后位于第 $k$ 个数是几。这是快速排序的一个经典应用。在快速排序的分块阶段完毕之后，小于等于 $piv$ 和大于等于 $piv$ 的部分被分在了两边，现在我们只关心排位第 $k$ 的数，所以我们可以根据 $k$ 在哪半边，来只去那半边寻找答案。

例题: [https://blog.csdn.net/qq\\_46105170/article/details/113794540](https://blog.csdn.net/qq_46105170/article/details/113794540)

递归写法:

```
1  #include <iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int n, k;
6  int a[N];
7
8  int quick_select(int l, int r, int k) {
9      int i = l, j = r;
10     int piv = a[l + (r - l >> 1)];
11     while (i <= j) {
12         while (a[i] < piv) i++;
13         while (a[j] > piv) j--;
14         if (i <= j) swap(a[i++], a[j--]);
15     }
16
17     if (k <= j) return quick_select(l, j, k);
18     if (k >= i) return quick_select(i, r, k);
19     // 如果有中间部分存在, 那么这个中间部分只会有一个数, 而且这个数就是答案
20     return a[k];
21 }
22
23 int main() {
24     cin >> n >> k;
25     for (int i = 0; i < n; i++) scanf("%d", &a[i]);
26
27     cout << quick_select(0, n - 1, k - 1) << endl;
28 }
```

非递归写法:

```
1  #include <iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int n, k;
6  int a[N];
7
8  int quick_select(int l, int r, int k) {
9     while (l < r) {
10         int i = l, j = r;
11         int piv = a[l + (r - l >> 1)];
12         while (i <= j) {
13             while (a[i] < piv) i++;
```

```

14     while (a[j] > piv) j--;
15     if (i <= j) swap(a[i++], a[j--]);
16 }
17
18     if (k <= j) r = j;
19     else if (k >= i) l = i;
20     else break;
21 }
22 return a[k];
23 }
24
25 int main() {
26     cin >> n >> k;
27     for (int i = 0; i < n; i++) scanf("%d", &a[i]);
28     cout << quick_select(0, n - 1, k - 1) << endl;
29 }

```

快速选择算法的平均时间复杂度是 $O(n)$ ，最坏 $O(n^2)$ ，递归写法平均空间复杂度 $O(\log n)$ ，最坏 $O(n)$ ，循环写法空间复杂度 $O(1)$ 。由于循环写法完全优于递归写法，所以推荐用循环写法。

注意到快速排序里我们是递归的写法，但快速选择里，我们却可以化为非递归的while循环的形式。这里的细节涉及到一个概念，叫“尾递归”，即 `tail recursion`。它指的是一个递归函数里，递归的那一步恰好只出现在整个函数的逻辑的最后。在这种情况下，递归函数很容易写为循环的形式。例如我们观察快速选择的递归写法，递归恰好只出现在最后一步。于是我们就想到，可以用循环，循环结束的条件就是递归出口，然后在最后一步的时候，将相关参数做相应修改。可以仔细体会下面两个写法，如何对应起来，体会怎么将尾递归的代码改为循环形式。值得一提的是，现在的有的编译器已经可以强大到，直接将尾递归的代码转换为循环形式来执行，以减少开销。

```

1  int quick_select(int l, int r, int k) {
2      int i = l, j = r;
3      // ...
4
5      if (k <= j) return quick_select(l, j, k);
6      if (k >= i) return quick_select(i, r, k);
7      return a[k];
8  }

```

```

1  int quick_select(int l, int r, int k) {
2      while (l < r) {
3          int i = l, j = r;
4          // ...
5          if (k <= j) r = j;
6          else if (k >= i) l = i;
7          else break;
8      }
9      return a[k];
10 }

```

现在我们回头来看快速排序的代码，其最后一句是

```

1  quick_sort(l, j), quick_sort(i, r);

```

即递归语句出现在了最后两句，从而不是尾递归，那么就不能改为循环实现。但是，我们仍然可以用栈来模拟递归。看下面的代码：

```

1  #include <iostream>
2  #include <stack>
3  #include <tuple>
4  using namespace std;
5  using PII = pair<int, int>;
6
7  const int N = 100010;
8  int a[N];
9
10 void quick_sort(int l, int r) {
11     stack<PII> stk;
12     stk.push({l, r});
13     while (stk.size()) {
14         tie(l, r) = stk.top(); stk.pop();
15         if (l >= r) continue;
16
17         int piv = a[l + (r - l >> 1)];
18         int i = l, j = r;
19         while (i <= j) {
20             while (a[i] < piv) i++;
21             while (a[j] > piv) j--;
22             if (i <= j) swap(a[i++], a[j--]);
23         }
24
25         stk.push({l, j}), stk.push({i, r});
26     }
27 }

```

```

28
29 int main() {
30     int n;
31     scanf("%d", &n);
32     for (int i = 0; i < n; i++) scanf("%d", &a[i]);
33
34     quick_sort(0, n - 1);
35     for (int i = 0; i < n; i++) printf("%d ", a[i]);
36 }

```

尝试比较这两段以体会区别：

```

1 void quick_sort(int l, int r) {
2     if (l >= r) return;
3
4     int piv = a[l + (r - l >> 1)];
5     int i = l, j = r;
6     while (i <= j) {
7         while (a[i] < piv) i++;
8         while (a[j] > piv) j--;
9         if (i <= j) swap(a[i++], a[j--]);
10    }
11
12    quick_sort(l, j), quick_sort(i, r);
13 }
14
15 void quick_sort(int l, int r) {
16     stack<PII> stk;
17     stk.push({l, r});
18     while (stk.size()) {
19         tie(l, r) = stk.top(); stk.pop();
20         if (l >= r) continue;
21
22         int piv = a[l + (r - l >> 1)];
23         int i = l, j = r;
24         while (i <= j) {
25             while (a[i] < piv) i++;
26             while (a[j] > piv) j--;
27             if (i <= j) swap(a[i++], a[j--]);
28         }
29
30         stk.push({l, j}), stk.push({i, r});
31     }
32 }

```

如果我们把第二种方法里的栈想象成系统调用栈，那么就能理解，系统调用栈和我们自己开与一个栈，本质上其实没有区别。

## 作业

---

上面的两道例题，快速排序和快速选择。其中快速排序要用递归和栈模拟递归两种方式来写，快速选择只要求掌握循环写法。

理解不同复杂度的含义；牢记上面算法的复杂度。

体会分治的思想，体会含递归的程序是怎么运行的，它的运行和栈的关系。