排序算法与分治2

排序算法的稳定性

如果一个排序算法不改变相等元素的相对位置，那么就说这个排序算法是稳定的。例如对于三个人A, B, C，它们的年龄是$30, 20, 20$，现在要将他们三个人按年龄排序，那么答案可以是BCA也可以是CBA。如果在排序之前，他们的顺序是ABC，即B在C前面，排完序之后依然如此，对应的就是BCA这种排序结果，那么这样的排序就是稳定的。

之前所介绍的快速排序是不稳定的，即它不能保证相同元素的相对位置不变。思考：我们能否通过某种手法，将之前快速排序的代码变为稳定？（可以将元素和下标做成pair之后再排序）

本文将要介绍的归并排序则是一个稳定的排序算法。

归并排序

归并排序是基于这样一种思想：

1. 先将整个数组平均分为两部分，然后直接递归排序这两部分。当然如果要排序的部分只含1个数，则不用排序了；
2. 将两部分各自排好序之后，利用双指针的手法，将两个有序数组合并为一个。可以这样做，首先开一个临时的数组，然后开两个指针分别指向两部分，从左往右扫，每次将较小的数填到临时数组里，直到两部分都被扫完。最后将临时数组回填到原数组之中。

例题：https://blog.csdn.net/qq_46105170/article/details/113792922

递归写法：

```cpp
#include <iostream>
using namespace std;

const int N = 100010;
int n;
int a[N], tmp[N];

void merge_sort(int l, int r) {
  if (l >= r) return;

  int m = l + (r - l >> 1);
  merge_sort(l, m), merge_sort(m + 1, r);

  // 双指针合并两个有序数组
  int k = l, i = l, j = m + 1;
```

```
  while (i <= m && j <= r) {
    // 取<=可以保证稳定性，因为等于的时候，左边的数依然排在左边
    if (a[i] <= a[j]) tmp[k++] = a[i++];
    else tmp[k++] = a[j++];
  }

  while (i <= m) tmp[k++] = a[i++];
  while (j <= r) tmp[k++] = a[j++];

  for (i = l; i <= r; i++) a[i] = tmp[i];
}

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) scanf("%d", &a[i]);

  merge_sort(0, n - 1);
  for (int i = 0; i < n; i++) printf("%d ", a[i]);
}
```

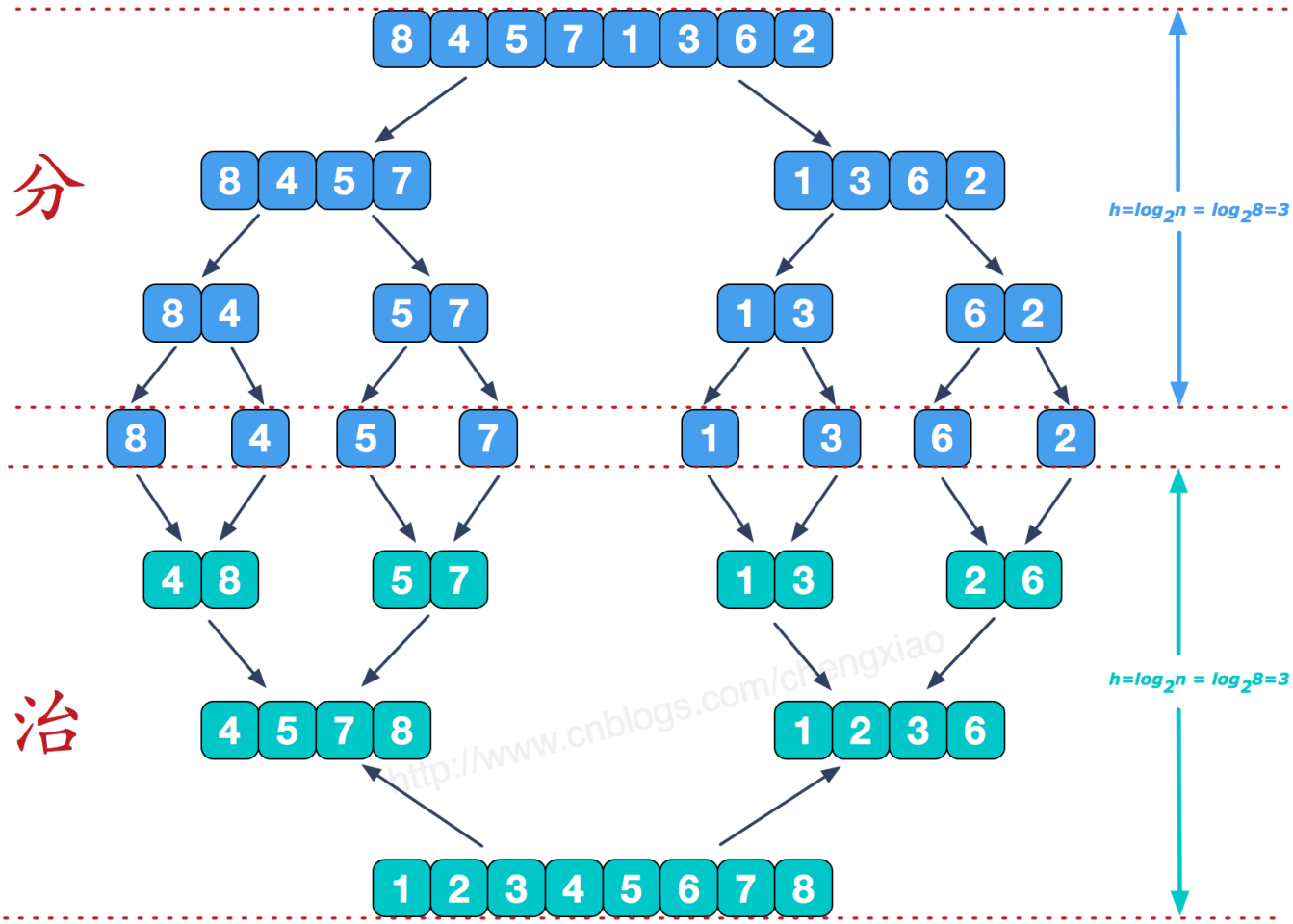
设归并排序对于长n数组消耗的时间是$T(n)$，由于合并部分消耗时间是$O(n)$，所以有

$$T(n) = 2T(\frac{n}{2}) + O(n) = 2(2T(\frac{n}{4}) + O(\frac{n}{2})) + O(n) = 4T(\frac{n}{4}) + O(n) + O(n) = \ldots = (\log_2 n)O(n)$$

可以推导出$T(n) = O(n \log n)$。如果想深入了解怎么从递推式求解算法复杂度，可以参考Master定理：https://zh.wikipedia.org/wiki/%E4%B8%BB%E5%AE%9A%E7%90%86。

空间复杂度$O(n)$，需要用到额外数组。

这张图很好的解释了分治算法是怎么做的：



按上面这张图的思想，我们可以将其写成非递归形式。我们先将整个数组两个两个分组，由于单个数是自动排好序的，所以2个元素的小组可以直接合并。例如对于数组$[8,4,5,7,1,3,6,2]$，两个两个分组成为：

$[[8,4],[5,7],[1,3],[6,2]]$

然后小组内部可以直接做双指针的“归并”，得到：

$[[4,8],[5,7],[1,3],[3,6]]$

接着我们考虑整个数组四个四个分组：

$[[4,8,5,7],[1,3,3,6]]$

我们发现每个小组内部，左半部分是排好序的，右半部分也是排好序的，从而又可以做归并了，可以得到：

$[[4,5,7,8],[1,3,3,6]]$

最后类似的，做最后一次归并，就可以将整个数组排好序。

这种思想写成代码即为：

```cpp
#include <iostream>
using namespace std;

const int N = 1e5 + 10;
int n;
int a[N], tmp[N];

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) scanf("%d", &a[i]);

  // k是每次归并的时候，左右两半部分的元素个数
  for (int k = 1; k < n; k <<= 1)
    // l是归并的时候左半部分的左端点
    for (int l = 0; l + k < n; l += k << 1) {
      int i = l, j = l + k, idx = l, r = min(l + (k << 1) - 1, n - 1);
      while (i < l + k && j <= r) {
        if (a[i] <= a[j]) tmp[idx++] = a[i++];
        else tmp[idx++] = a[j++];
      }

      while (i < l + k) tmp[idx++] = a[i++];
      while (j <= r) tmp[idx++] = a[j++];

      for (idx = l; idx <= r; idx++) a[idx] = tmp[idx];
    }

  for (int i = 0; i < n; i++) printf("%d ", a[i]);
}
```

可以不可以开手动栈模拟递归的写法来写呢？当然是可以的，但是上面的写法连栈都不用开，显然更优。

求逆序对个数

一个序列a里的逆序对指的是$\{(i,j)|i<j,a[i]>a[j]\}$。利用归并排序可以求逆序对的个数。我们可以用分治的思想来考虑，首先将数组平均分为两半，然后直接递归求解两边内部的逆序对个数；接着只需要考虑跨越中线的逆序对个数即可。如果两半边都已经排好序，那么求跨中线的逆序对的个数是可以用双指针来做的。

例题：https://blog.csdn.net/qq_46105170/article/details/113794612

例如：$[\mathbf{1},5,7,\mathbf{2},4,6]$

一开始两个指针i,j分别指向两半部分的第一个数$1,2$，设右半部分的起始位置是s，即一开始$j=s$，接着做比较。我们对每个j，求最小的i使得$a[i]>a[j]$，由于两半部分都是排好序的，所以我们知道$a[i:s-1]$都是大于$a[j]$的，从而$a[j]$组成的逆序对个数为$s-i$。对于不同的i求解对应的j就能知道总共的跨中线的逆序对个数了。

注意，求完当前子数组跨中线的逆序对个数的同时，我们也要顺便对其进行排序（手法和归并排序是类似的，二路归并，小的数填进tmp数组最后回填），这样递归回上层的时候，两半部分是有序的，然后又可以同样方法求跨中线逆序对个数了。代码如下：

```cpp
#include <iostream>
using namespace std;
using ll = long long;

const int N = 100010;
int n;
int a[N], tmp[N];

ll merge_sort(int l, int r) {
  if (l >= r) return 0;

  int m = l + (r - l >> 1);
  ll res = merge_sort(l, m) + merge_sort(m + 1, r);

  int i = l, j = m + 1, idx = l;
  while (i <= m && j <= r) {
    if (a[i] > a[j]) {
      res += m - i + 1;
      tmp[idx++] = a[j++];
    } else
      tmp[idx++] = a[i++];
  }

  while (i <= m) tmp[idx++] = a[i++];
  while (j <= r) tmp[idx++] = a[j++];

  for (i = l; i <= r; i++) a[i] = tmp[i];
  return res;
}

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) scanf("%d", &a[i]);

  cout << merge_sort(0, n - 1) << endl;
}
```

作业

上面所有例题。归并排序要用递归和非递归两种写法写。

体会分治思想在归并排序和求逆序对个数当中的作用。