排序

dbywsc

2025/7

目录

定义

排序算法(Sorting algorithm) 是一种将一组特定的数据按某种顺序进行排列的算法。排序算法多种多样，性质也大多不同。

性质 – I

我们通常用下面几种性质描述一个排序算法：
稳定性：
稳定性是指相等的元素经过排序之后相对顺序是否发生了改变。拥有稳定性这一特性的算法会让原本有相等键值的纪录维持相对次序，即如果一个排序算法是稳定的，当有两个相等键值的纪录 R 和 S，且在原本的列表中 R 出现在 S 之前，在排序过的列表中 R 也将会是在 S 之前。

性质 – II

时间复杂度：
时间复杂度用来衡量一个算法的运行时间和输入规模的关系，通常用 O 表示。
简单计算复杂度的方法一般是统计「简单操作」的执行次数，有时候也可以直接数循环的层数来近似估计。
时间复杂度分为最优时间复杂度、平均时间复杂度和最坏时间复杂度。算法竞赛中要考虑的一般是最坏时间复杂度，因为它代表的是算法运行水平的下界，在评测中不会出现更差的结果了。
基于比较的排序算法的时间复杂度下限是 $O(n \log n)$ 的。
空间复杂度：
与时间复杂度类似，空间复杂度用来描述算法空间消耗的规模。一般来说，空间复杂度越小，算法越好。

表: 常见排序算法的时间／空间复杂度及稳定性

算法	最好情况	平均情况	最坏情况	空间复杂度	稳定性
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	是
基数排序	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	是
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	否
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	是
桶排序	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n+k)$	视实现而定
希尔排序	$O(n\log n)$	$O(n(\log n)^2)$	$O(n^2)$	$O(1)$	否
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	否

在下文中，我们将详细介绍快速排序和归并排序的原理和写法。

定义

归并排序（merge_sort) 是高效的基于比较的稳定排序算法。
性质:
最优、最坏、平均时间复杂度均为 $O(n \log n)$ ，空间复杂度为 $O(n)$ ，是 **稳定** 的排序算法。

原理

在本节中，我们介绍基于**分治**思想的归并排序。
归并排序的过程可以分成三步：
1. 分解：将待排序的数组从中间分成两部分，如果此时数组的长度为 1 ，一定是有序的，所以不再分解。
2. 解决：检查分解出来的数组是否有序（用第一步），如果有序，那么将它们合并（第三步）为一个有序的数组，否则对不有序的的数组重复第二条（即继续分解-> 判断-> 如果有序合并），直到有序。
3. 合并：将有序的子数组合并为一个大的有序数组。

实现

```
int tmp[N];
void merge_sort(int arr[], int l, int r) {
    if(l >= r) return;
    int mid = (l + r) / 2;
    merge_sort(arr, l, mid); merge_sort(arr, mid + 1, r);
    int k = 1, i = l, j = mid + 1;
    while(i <= mid && j <= r) {
        if(arr[i] <= arr[j]) tmp[k++] = arr[i++];
        else tmp[k++] = arr[j++];
    }
    while(i <= mid) tmp[k++] = arr[i++];
    while(j <= r) tmp[k++] = arr[j++];
    for(int i = l, j = 1; i <= r; i++, j++) arr[i] = tmp[j];
}
```

不妨在 P1177 【模板】排序练习一下这道题。

应用 – I

归并排序的一个常用用法是求逆序对。

P1908 逆序对

由于归并排序每次递归会将数组分成左右两个部分，左边的数组优先进行排序。此时我们对左边的数组做一个遍历，将它们与右边的数组比较。如果当前的元素和右边的某个位置是逆序对，由于左边是有序的，因此对于左边这部分数组，从当前位置开始直到结束的所有元素都和右边是逆序对，因此会产生 $mid - i + 1$ 的贡献。

应用 – II

```
void merge_sort(int arr[], int l, int r) {
    if(l >= r) return;
    int mid = (l + r) / 2;
    merge_sort(arr, l, mid); merge_sort(arr, mid + 1, r);
    int k = 1, i = l, j = mid + 1;
    while(i <= mid && j <= r) {
        if(arr[i] <= arr[j]) tmp[k++] = arr[i++];
        else tmp[k++] = arr[j++], ans += mid - i + 1;
    }
    while(i <= mid) tmp[k++] = arr[i++];
    while(j <= r) tmp[k++] = arr[j++];
    for(int i = l, j = 1; i <= r; i++, j++) arr[i] = tmp[j];
}
```

STL 中的归并排序

事实上，在 STL 库中提供了归并排序函数 *stable_sort*() 作为一种稳定排序算法，它的用法与 *sort*() 一致：
stable_sort(*a.begin*(), *a.end*(), *cmp*) 其中 *cmp* 是自定义排序规则。

定义

快速排序 (quick_sort) 是一种被广泛应用的排序算法。
最优时间复杂度 $O(n \log n)$ ，最坏时间复杂度 $O(n^2)$ ，平均时间复杂度 $O(n \log n)$。
空间复杂度 $O(\log n)$
是 **不稳定** 的排序算法。

原理

与归并排序一样，快速排序同样使用了**分治**的思想。快速排序的过程如下：1. 分解：对待排序的数组 $A[p..r]$ 选择一个基准点[1] q，将 $A[p..r]$ 划分为两个子数组（可能为空）$A[p..q-1]$ 和 $A[q+1..r]$ ，使得 $A[p..q-1]$ 中的每一个元素都小于等于 $A[q]$ ，$A[q+1..r]$ 中的每一个元素都大于等于 $A[q]$ 。
2. 解决：通过递归调用快速排序，对子数组 $A[p..q-1]$ 和 $A[q+1..r]$ 也进行步骤一。
与归并排序不同的是，由于我们是直接对原数组进行了修改，因此不需要合并。

[1] 关于基准点的选择，通常有两种做法。一种方法是随机的选择一个位置，这种做法可以保证快速排序的时间复杂度；另一种方法先线性的找出当前数组的**中位数**，再以中位数作为基准点。这里我们介绍第一种写法。

优化 – I

由于算法竞赛中常常需要考虑算法的时间复杂度下限，因此朴素快速排序的最坏时间复杂度 $O(n^2)$ 无法满足我们的要求，在这里介绍一种快速排序的优化方法。

优化 – II

在选取基准点后，将待排序数组划分为三个部分：小于基准点、等于基准点、大于基准点。这样做相当于直接将等于基准点的元素聚集在基准点周围，跳过了继续递归的过程。
这样优化过的快速排序名为 **三路快速排序** ，本质上是基数排序和快速排序的融合。它将最优时间复杂度优化到了 $O(n)$ ，并且降低了时间复杂度退化到 $O(n^2)$ 的可能。

实现

```
void quick_sort(int arr[], int len) {
    if(len <= 1) return;
    int pivot = arr[rand() % len];
    int i = 0, j = 0, k = len;
    while(i < k) {
        if(arr[i] < pivot) std::swap(arr[i++], arr[j++]);
        else if(pivot < arr[i]) std::swap(arr[i], arr[--k]);
        else i++;
    }
    quick_sort(arr, j); quick_sort(arr + k, len - k);
}
```

不妨在 P1177 【模板】排序练习一下这道题。

STL 中的快速排序

事实上，STL 库中的 $sort()$ 是使用了另一种优化方式的排序算法，名为**内省排序**，本质上是快速排序和 **堆排序** 的结合，**严格保证**了最坏时间复杂度为 $O(n \log n)$ ，它的用法我们已经见过了：$sort(a.begin(), a.end(), cmp)$ ，其中 cmp 是自定义排序规则。

排序算法无论在算法竞赛中，还是在实际开发中都是非常常用的算法，一般情况下 *sort*() 的性能已经能够满足我们的需求，但是有时，我们要根据题目数据的特点和需求的不同选择不同的排序算法。

P1309 [NOIP 2011 普及组] 瑞士轮

P1012 [NOIP 1998 提高组] 拼数

P1104 生日