排序

快速排序 & 归并排序 & 堆排序 etc.

邵逸帆 $q\omega q$

23 电信基地班
兰州大学算法与程序设计集训队

2024 年 7 月 18 日

目录

Intro

- 排序是将一组数据按照某种顺序重新排列的过程。
- 稳定性: 若两个相等的元素在排序前后的相对位置不变，则称排序算法是稳定的。
- 时间复杂度：简单计算复杂度一般是通过统计“简单操作”的次数来实现的。基于比较的排序算法的时间复杂度的下界是 $O(n\log n)$。
- 空间复杂度：排序算法的空间复杂度是指除了输入数据外，算法运行时所需的额外空间。

通常排序算法可以分为三类：

1. 冒泡排序、选择排序、插入排序
2. 快速排序、归并排序、堆排序
3. 计数排序、桶排序、基数排序

前两类是基于比较的排序，第一类的时间复杂度是 $O(n^2)$，第二类的时间复杂度是 $O(n\log n)$。其中快速排序由于其高效性被广泛使用, 而归并排序由于其稳定性被用于外部排序。

第三类换了一种思路，不直接比较元素的大小，而是对被排序的元素采取按位划分、分类映射的方法，可以实现线性时间复杂度。

八大排序	时间复杂度	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{\frac{3}{2}})$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(1)$	不稳定
计数排序	$O(n+k)$	$O(k)$	稳定

分治即“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

分治主要包含三个步骤：

- 分解：将原问题分解成一系列子问题。
- 解决：递归地解决这些子问题。
- 合并：将子问题的解合并成原问题的解。

简单来说就是递归前要做什么 (分解)，递归后要做什么 (合并)。

归并排序和快速排序都是基于分治思想的排序算法。

快速排序

选择一个基准元素 pivot，将数组分成两部分，左边的元素都小于等于 pivot，右边的元素都大于等于 pivot。递归地对左右两部分做快速排序。

划分

- 选择一个基准元素 pivot。
- 用两个指针 $p1, p2$ 分别指向数组的起始位置和结束位置。
- 从 $p1$ 开始向右找到第一个大于等于 pivot 的元素，从 $p2$ 开始向左找到第一个小于等于 pivot 的元素，交换这两个元素。
- 重复上述过程直到 $p1$ 和 $p2$ 相遇。
- 将 pivot 与 $p1$ 指向的元素交换。

```
void quick_sort(int l, int r) {
  if (l >= r) return;
  int i = l - 1, j = r + 1;
  int x = a[rand() % (r - l + 1) + l]; // 随机选择基准
  while (i < j) {
    do i++; while (a[i] < x);
    do j--; while (a[j] > x);
    if (i < j) swap(a[i], a[j]);
  } // 此时a[l~j]都小于等于x,a[j+1~r]都大于等于x
  quick_sort(l, j), quick_sort(j + 1, r);
}
```

快速排序是一种原地排序算法，不需要额外的空间。但是快速排序是不稳定的，因为在划分的过程中可能会改变相同元素的相对位置。

快速排序的平均时间复杂度是 $O(n\log n)$，最坏情况下的时间复杂度是 $O(n^2)$，最坏情况发生在数组已经有序的情况下。

为了避免最坏情况的发生，我们可以随机选择基准元素，或者选择中位数作为基准元素。

利用快速排序的思想，我们可以在 $O(n)$ 的时间复杂度内找到数组中的第 k 大数 (或者第 k 小数)。

- 我们选择一个基准元素 x，将数组分成两部分，左边的元素都小于等于 x，右边的元素都大于等于 x。
- 如果左边的元素个数大于等于 k，那么我们在左边找第 k 大数。
- 否则我们在右边找第 $k - num$ 大数，其中 num 是左边元素的个数。

在最终求得答案时，我们并没有对所有的数组进行排序，而是利用了每一次划分的信息。

归并排序

- 分解：将数组分成两半。
- 解决：递归地对两半进行归并排序。
- 合并：将两个有序数组合并成一个有序数组。

合并有序数组

- 申请一个临时数组 tmp，大小为 n。
- 用两个指针 $p1, p2$ 分别指向两个有序数组的起始位置。
- 比较 $p1, p2$ 指向的元素，将较小的元素放入 tmp 中。
- 重复上述过程直到两个数组中的元素全部放入 tmp 中。
- 将 tmp 中的元素复制回原数组。

Merge Sort

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

归并排序是一种稳定的排序算法，时间复杂度是 $O(n\log n)$，空间复杂度是 $O(n)$。

归并排序的缺点是需要额外的空间，但是归并排序是一种天然适合外部排序的算法。

归并排序的优点是可以很容易地将其改造为并行算法，可以利用多核 CPU 或者多机器进行并行计算。

满足 $i < j$ 且 $a[i] > a[j]$ 的数对称为逆序对。在每次合并的时候，我们考虑左区间元素大于右区间元素的情况，此时左区间的剩余元素个数即为逆序对的个数。

```
void merge(int l, int r) {
  /*...*/ int i = l, j = mid + 1;
  for (int k = l; k <= r; k++) {
    if (j > r || (i <= mid && a[i] <= a[j])) {
      tmp[k] = a[i++];
    } else {
      // 此时a[i~mid]都大于a[j]
      tmp[k] = a[j++], ans += mid - i + 1;
    }
  } /*...*/
}
```

目录

我们可以使用堆来优化选择排序，这样选出一个最值的时间复杂度可以降低到 $O(\log n)$。

- 堆是一种数据结构。对于任意一个节点，其父节点的值大于等于 (或小于等于) 其子节点的值。
- 堆可以用数组来表示，对于节点 i，其左儿子为 $2i$，右儿子为 $2i+1$，父节点为 $i/2$。
- 堆分为大顶堆和小顶堆，大根堆的根节点是最大的元素，小根堆的根节点是最小的元素。
- 堆是一棵完全二叉树，每一个节点的子树都是一个堆。

堆排序

- 建堆：将数组构建成一个大顶堆。可以证明其为 $O(n)$。
- 排序：将堆顶元素与最后一个元素交换，然后调整堆。

调整堆

- 从根节点开始比较左右子节点的值，将较大的子节点与根节点交换。
- 递归地对交换后的子节点进行调整。

堆排序本质上是一种选择排序。从调整堆的操作中可以看出，堆排序涉及到较远项的交换，从而是不稳定的。

伪代码

Algorithm 1 Heap Sort

```
BuildHeap()
for i = n to 2 do
    Swap(a[1], a[i])
    AdjustHeap(1, i - 1)
end for
```

Algorithm 2 Build Heap

```
for i = n/2 to 1 do {where i is non-leaf node}
    AdjustHeap(i, n){调整 i 为根的堆}
end for
```

伪代码

Algorithm 3 Adjust Heap

$t = a[i]$ {下沉 i 节点}
for $j = 2 \times i; j \leq n; j = 2 \times i$ **do**
    if $j < n$ and $a[j] < a[j+1]$ **then** {选择左右儿子中的较大的}
        $j{+}{+}$
    end if
    if $t \geq a[j]$ **then** {满足堆的性质就退出}
        break
    end if
    $a[i] = a[j]$ {上浮 j 节点}
    $i = j$
end for
$a[i] = t$

在我们学习了堆这一数据结构之后。我们可以使用对顶堆的思想来维护一个动态中位数。

即我们维护两个堆，一个大根堆，一个小根堆。大根堆存储较小的一半元素，小根堆存储较大的一半元素。

不过这在之前已经提到了，就不再赘述。

兰州大学

void sort(RandomIt first, RandomIt last, Compare comp);

这是 std::sort 的函数原型。其中 RandomIt 是一个随机访问迭代器，排序区间是左闭右开的；Compare 是一个可调用对象，用于比较待排序的元素，默认为 std::less<T>()。

STL 中的 std::sort 函数是非常高效的

- C++ STL 中的 sort 函数是基于快速排序的。
- 对于小规模数据，sort 函数会使用插入排序。
- 对于大规模数据，sort 函数会使用快速排序。
- 对于近乎有序的数据，sort 函数会使用三路快速排序。

例如

```
int a[1000000], n; // 对于数组a进行排序
std::sort(a, a + n); // 默认升序排序
std::sort(a, a + n, std::greater<int>()); // 降序排序

std::vector<int> v; // 对于vector v进行排序
std::sort(v.begin(), v.end());
```

对于 std::pair 和 std::tuple 这些已经实现比较的类型也可以直接排序。

当然，STL 还提供了诸如 std::stable_sort, std::partial_sort, std::nth_element, std::is_sorted 等函数。

对于自定义类型，我们可以重载其比较运算符，或者使用 lambda 表达式来进行排序。

重载 operator<

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

lambda 表达式

```
std::sort(v.begin(), v.end(),
  [](const Node &a, const Node &b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
});
```

当然一些可调用对象也是可以的，通常它们被称为仿函数 (functor)。

在这里我们需要注意的是：我们自定义的比较函数需要满足“严格弱序”，这是 STL 所要求的。

严格弱序的定义如下：

- 反自反性：$!comp(a, a)$
- 反对称性：$comp(a, b) \Rightarrow !comp(b, a)$
- 传递性：$comp(a, b)\ \&\&\ comp(b, c) \Rightarrow comp(a, c)$

定义 $equiv(a, b)$ 为 $!comp(a, b)\ \&\&\ !comp(b, a)$, 若满足自反性, 对称性, 传递性, 那么 $comp$ 在 $equiv$ 所确定的等价类上引入了一种严格全序。

假如我们定义 < 为 <=，那么我们就不满足严格弱序的定义，使用 std::sort 会出现未定义行为，可能会 Runtime Error。

去重

对于一个有序的数组，我们很容易对其进行去重。当然 STL 也提供了对应的函数，即 std::unique。

去重

```
std::vector<int> v;
std::sort(v.begin(), v.end());
v.erase(std::unique(v.begin(), v.end()), v.end());
// 这样就完成了对于数组v的去重操作。
```

std::unique 函数会将重复的元素放到数组的末尾，并返回一个指向第一个重复元素的迭代器，我们可以通过 erase 函数来删除这些重复元素。

离散化

离散化是一种将数据映射到连续的整数区间的方法。在一些问题中，我们需要将一些离散的数据映射到连续的整数区间，以便于我们进行操作。

- 离散化的过程是将所有的数据放入一个数组中，然后对数组进行排序。
- 然后对于每一个数据，我们可以通过二分查找找到其在排序后的数组中的位置。
- 通过这种方法，我们可以将数据映射到 $[1, n]$ 的整数区间。

```
void discrete() {
  std::sort(a + 1, a + n + 1);
  m = std::unique(a + 1, a + n + 1) - a - 1;
}
int query(int x) {
  return std::lower_bound(a + 1, a + m + 1, x) - a;
}
```

THX 4 *Listening*! *:)*