数据结构与算法

第 7 讲：排序

韩文弢

清华大学

2024—2025 学年度春季学期

本讲内容

7.1 问题引入

7.1.1 姓名排序

在日常生活中，经常要对名单中的姓名进行排序。对于中文姓名，在会议名单、获奖名单等正式场合通常使用**按姓氏笔画为序**。

按姓氏笔画为序的简易规则如下：

- 对于姓氏以规范简体字的笔画数少的排序在前，如“丁”（2 画）排在“王”（4 画）前
- 笔画数相同的按笔顺[1]排序（横竖撇捺折），如“李”（7 画，首笔横）排在“吴”（7 画，首笔竖）前
- 若姓氏相同，继续比较名字，单名可视作名字的第一个字为 0 画

[1]详细规则参见《GB 13000.1 字符集汉字字序（笔画序）规范》

7.1.2 姓名排序示例

丰　云	王宏旭	王若水	王思涵	王彦淞	王海博
尤子仪	毛思静	石嘉木	史尚坤	包斯佳	刘春龙
刘宪武	米欣然	江　瑞	许呈睿	阮黄阳	李文逍
李卓文	李炆东	李俊超	李锦钊	李澳翔	杨再鹏
杨闰渊	杨哲焓	肖博洋	邱颖婷	余　璟	余卓磊
余亮阳	应季轩	张世麒	张宏达	张雅锟	张镕奎
陈　佑	陈　凯	陈奕宇	武樊浩	易沁韬	罗时琛
周　畅	周　涵	周泓羽	郑进威	屈子湉	封昊秩
赵乐毅	查恺迪	徐　宇	高西来	高轩博	郭亚菲
郭熙睿	唐锦松	谈子银	展　然	黄俊玮	黄叙霖
曹忠航	曹稷阳	龚跃东	梁勇卿	彭　锐	鲁明洋
谢　佳	翟彝凡	薛佳琪	霍子元	魏信韬	魏家新

7.2 排序的概念

7.2.1 排序的定义

定义 7.1 (排序) 将数据元素按照关键字重新排列为升序（从小到大）或降序（从大到小）的操作称为排序。

简便起见本课程中的排序默认以**升序**为例进行讲解。

例. 有平面上的点的坐标数据 $(1,0),(0,1),(-1,0),(0,-1)$，以 y 坐标为第一关键字，x 坐标为第二关键字，则排序结果为

$$(0,-1),(-1,0),(1,0),(0,1)$$

7.2.2 排序算法的分类

有很多种排序算法，可以根据不同的标准对排序算法进行分类：

1. 根据数据的存放位置：**内排序**指排序期间全部元素都存储在内存中，在内存中调整存放位置；**外排序**指元素存储在外存，排序过程中借助内存调整元素在外存的存放位置。
2. 根据排序的实现手段：**基于比较和交换的排序**指排序中的基本操作为元素的大小比较和位置交换，例如选择排序和堆排序；**基于分配的排序**指排序中的基本操作是把元素分配到不同的容器中。
3. 排序的**稳定性**：对于相等的元素，如果排序前后保持相对顺序不变，称为稳定的排序算法，否则称为不稳定的排序算法。

7.2.3 排序的稳定性举例

例. 对于平面上的点的坐标数据 $(1,0),(0,1),(-1,0),(0,-1)$，以 x 坐标为关键字，如果能保证每次排序的结果都是

$$(-1,0),(0,1),(0,-1),(1,0)$$

则排序算法是稳定的。

如果可能会出现

$$(-1,0),(0,-1),(0,1),(1,0)$$

则排序算法是不稳定的。

7.2.4 排序稳定性的用途

如果一种排序算法是稳定的，则可以通过以不同关键字多次调用该算法实现多关键字排序。

例. 对于平面上的点的坐标数据 $(1,0),(0,1),(-1,0),(0,-1)$，先以 x 坐标为关键字进行排序，得到

$$(-1,0),(0,1),(0,-1),(1,0)$$

再以 y 坐标为关键字进行排序，得到

$$(0,-1),(-1,0),(1,0),(0,1)$$

如此就完成了以 y 坐标为第一关键字，x 坐标为第二关键字的排序。

7.3 排序的实现

7.3.1 常见排序算法

- 选择排序（selection sort）：时间复杂度 $O(n^2)$，空间复杂度 $O(1)$
- 堆排序（heapsort）：时间复杂度 $O(n \log n)$，空间复杂度 $O(1)$
- 插入排序（insertion sort）
- 交换排序（exchange sort）
- 快速排序（quicksort）
- 归并排序（merge sort）
- 内省排序（introsort）
- Tim 排序（Timsort）
- ……

7.3.2 插入排序

考虑打扑克牌时的抓牌过程，手上维护有序的牌组，每次抓到一张牌后将其**插入**到手牌中。

受此启示，可设计插入排序算法：将整个要排序的序列看作已排序和待排序两部分。

1. 开始时已排序部分只有一个元素，其余元素都在待排序部分中。
2. 每次从待排序部分中拿出一个元素，插入到已排序部分的合适位置。
3. 当所有元素都插入到已排序部分后，排序完成。

7.3.3 插入排序算法

伪码 7.1: 插入排序

```
InsertionSort(A):
1   for i ← 1 to A.size − 1
2       x ← A[i]
3       j ← i
4       while j > 0 ∧ x < A[j − 1]
5           A[j] ← A[j − 1]
6           j ← j − 1
7       A[j] ← x
```

7.3.4 插入排序过程示例

位置	0	1	2	3	4	5	6	7
初始	4	1	5	7	3	8	6	2
第 1 趟	1	4	5	7	3	8	6	2
第 2 趟	1	4	5	7	3	8	6	2
第 3 趟	1	4	5	7	3	8	6	2
第 4 趟	1	3	4	5	7	8	6	2
第 5 趟	1	3	4	5	7	8	6	2
第 6 趟	1	3	4	5	6	7	8	2
第 7 趟	1	2	3	4	5	6	7	8

7.3.5 插入排序的时间复杂度分析

插入排序算法外层循环进行 $n-1$ 次迭代，对于内层循环：

1. 最好情况：满足 $A[i] \geq A[i-1]$，此时不执行内层循环体，比较 1 次，交换 0 次。
2. 最坏情况：满足 $A[i] < A[0]$，此时内层循环体执行 i 次，比较 i 次，交换 i 次。

因此，插入排序算法最好情况下的时间复杂度为 $O(n)$，最坏情况下的时间复杂度为 $O(n^2)$。什么是最好情况？什么是最坏情况？

插入排序算法平均情况下的时间复杂度为 $O(n^2)$。

7.3.6 插入排序的优缺点

插入排序的优点：

- 实现简单
- 对于小规模数据非常高效（常数小）
- **自适应性**：对于基本有序的数据，算法非常有效
- **稳定性**：能够保持相等元素的相对次序
- **就地**：只需要 $O(1)$ 的额外空间
- **在线**：能够增量地进行排序

缺点：对于大规模数据效率很低。

7.3.7 快速排序

插入排序每次都要将一个新的元素插入到**整个**已排序部分，是效率低下的原因。

考虑在序列内进行调整，使得序列能尽快呈现有序的样子：用序列中的某个元素（称为轴点，pivot）对序列进行**划分**，使得比轴点小的元素都在轴点前面，比轴点大的元素都在轴点后面。

此时，序列中的元素除轴点外被轴点分为前后两个部分，可分别**递归**进行划分。由于划分后的子序列长度严格变短，而长度为 0 或 1 的序列平凡有序，因此递归总会终止。

划分有多种实现方式，这里介绍比较简单的一种。

7.3.8 序列划分算法

伪码 7.2: 序列划分

```
Partition(A):
1   m ← A.size
2   p ← A[m − 1]   # 以最后一个元素为轴点进行划分
3   i ← 0   # 位置 i 之前都是小于轴点的元素
4   for j ← 0 to m − 2
5       if A[j] < p   # 将比轴点小的元素交换至位置 i 之前
6           A[i] ↔ A[j]
7           i ← i + 1
8   A[i] ↔ A[m − 1]
9   return i
```

7.3.9 快速排序算法

令 $A[i:j]$ 表示顺序表 A 中从第 i 个元素到第 j 个元素（半闭半开）构成的子表，与原来的顺序表共享相同的存储空间。

伪码 7.3: 快速排序

```
Quicksort(A):
1   n ← A.size
2   if n > 1
3       k ← Partition(A)
4       Quicksort(A[0 : k])
5       Quicksort(A[k + 1 : n])
```

7.3.10 快速排序过程示例

位置	0	1	2	3	4	5	6	7
初始	4	1	5	7	3	8	6	2
第 1 趟划分	1	2	5	7	3	8	6	4
第 2 趟划分	1	2	3	4	5	8	6	7
第 3 趟划分	1	2	3	4	5	6	7	8
第 4 趟划分	1	2	3	4	5	6	7	8

7.3.11 快速排序算法的时间复杂度分析

划分算法的时间复杂度为 $O(m)$。对于快速排序算法，

最坏情况：每次轴点取到的都是最大元素，划分后一边没有元素，另一边的元素个数是原来的个数减一，问题变为 $n-1$ 规模。此时，时间复杂度为 $O(n^2)$，空间复杂度为 $O(n)$。

最好情况：每次轴点取到的都是中位数，划分后两边长度相等，问题变为两个 $\frac{n}{2}$。此时，时间复杂度为 $O(n\log n)$，空间复杂度为 $O(\log n)$。

平均情况下，快速排序算法的时间复杂度为 $O(n\log n)$，空间复杂度为 $O(\log n)$。

7.3.12 轴点的选取

快速排序算法的效率与轴点的选取密切相关，为了避免退化情况，可以考虑：

- 在首、中、末三个元素中选中位数
- 随机选取一个元素

注意上述两种改进方法仍然是确定性的（计算机中的随机是用伪随机算法产生的），因此可以构造特定的序列，使得快速排序算法仍然会退化成 $O(n^2)$。

退化的情况会让快速排序运行时间特别长，可以作为拒绝服务攻击手段。

7.3.13 内省排序

在所有基于比较的排序中，快速排序是平均运行速度最快的排序方法。然而快速排序有其劣势，在某些情况下的表现不如其他排序算法。

为了提升排序算法的综合效率，可以根据不同情况考虑使用多种排序算法。

内省排序算法是一种混合排序算法，其思想是：先使用快速排序，当递归超过一定深度后转为堆排序；同时，如果元素个数过少，则会采用插入排序算法。

7.3.14 内省排序算法

伪码 7.4: 内省排序

```
Introsort(A, d):
1   n ← A.size
2   if n ≤ 小数据阈值
3       InsertionSort(A)
4   elseif d = 0
5       Heapsort(A)
6   else
7       k ← Partition(A)
8       Introsort(A[0 : k], d − 1)
9       Introsort(A[k + 1 : n], d − 1)
```

7.3.15 C++ 中的排序相关算法

函数	作用	时间复杂度
`sort`	将范围按升序排序，通常使用内省排序实现	$O(n \log n)$
`sort_heap`	将建成最大堆的范围按升序排序	$O(n \log n)$
`stable_sort`	将范围按升序排序，保持相等元素的顺序	$O(n \log n)^*$ 或 $O(n \log^2 n)$
`is_sorted`	检查范围是否已按升序排列	$O(n)$
`is_sorted_until`	找出从头开始的最长有序子范围	$O(n)$

*额外内存足够时

7.3.16 sort 使用示例

C++

```cpp
#include <algorithm>
#include <array>
#include <functional>
#include <iostream>
#include <string_view>

int main() {
  std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

  auto print = [&s](std::string_view const rem) {
    for (auto a : s) std::cout << a << ' ';
    std::cout << ": " << rem << '\n';
  };

```

7.3.16 sort 使用示例

```
15    std::sort(s.begin(), s.end());
16    print("用默认的 operator< 排序");
17
18    std::sort(s.begin(), s.end(), std::greater<int>());
19    print("用标准库比较函数对象排序");
20
21    struct {
22      bool operator()(int a, int b) const { return a < b; }
23    } customLess;
24
25    std::sort(s.begin(), s.end(), customLess);
26    print("用自定义函数对象排序");
27
```

7.3.16 sort 使用示例

```
28    std::sort(s.begin(), s.end(), [](int a, int b) { return a > b; });
29    print("用 lambda 表达式排序");
30  }
```

7.4 小结

7.4.1 算法设计思想

插入排序采用**增量**思想，使用**迭代**方式来实现。可以使用归纳法来证明迭代式算法的正确性或求解复杂度等性质。

快速排序采用**分而治之**（也叫分治，divide and conquer）的思想，使用**递归**方式来实现。一般来说可以把分治算法分为分、治、合三个步骤。

单一算法不能满足所有情况时，可以综合采用多种算法协同来解决问题，形成**混合**算法。

7.4.2 常见排序算法的比较

算法	时间复杂度			空间复杂度	稳定性
	最好	平均	最坏		
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	否
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)^*$	否
内省排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(\log n)$	否

*最坏情况下