杭州電子科技大學
HANGZHOU DIANZI UNIVERSITY

# 程序设计实践报告

## 基于 C 语言的排序算法设计以及效率分析

卓越学院
计算机科学英才班
周靖凯

# 目录

# 1 问题描述

排序是计算机科学中最基本和最重要的问题之一。它涉及将一组数据元素按照特定的顺序（通常是升序或降序）重新排列。有效的排序对于数据处理和信息检索是至关重要的，因为它大大提高了数据检索的效率。在本报告中，我们将探讨和评估几种常见的排序算法，包括冒泡排序、选择排序、插入排序、归并排序、快速排序以及其他特殊排序算法。

排序算法的效率和适用性取决于多种因素，如算法的时间复杂度、空间复杂度、稳定性，以及数据的初始状态。在实际应用中，选择合适的排序算法需要综合考虑这些因素。本报告旨在通过对不同排序算法在不同数据规模（$10^3$、$10^4$、$10^5$、$10^6$、$10^7$ ）和不同数据状态（完全无序和几乎有序）下的性能评估，为实际应用中的算法选择提供指导。

我们将使用标准化的测试方法来评估每种算法的性能，包括它们的执行时间和操作效率。此外，我们还将考察这些算法在几乎有序的数据集上的表现，以了解它们在处理部分有序数据时的效率。通过这些分析，我们期望能够更好地理解每种排序算法的优势和局限，为实际应用中的算法选择和优化提供坚实的理论基础。

# 2 主要算法

## 2.1 冒泡排序

### 2.1.1 定义

冒泡排序（英语：Bubble sort）是一种简单的排序算法。由于在算法的执行过程中，较小的元素像是气泡般慢慢「浮」到数列的顶端，故叫做冒泡排序。

### 2.1.2 过程

它的工作原理是每次检查相邻两个元素，如果前面的元素与后面的元素满足给定的排序条件，就将相邻两个元素交换。当没有相邻的元素需要交换时，排序就完成了。

经过 $i$ 次扫描后，数列的末尾 $i$ 项必然是最大的 $i$ 项，因此冒泡排序最多需要扫描 $n-1$ 遍数组就能完成排序。

### 2.1.3 稳定性

冒泡排序是一种稳定的排序算法。

### 2.1.4 时间复杂度

在序列完全有序时，冒泡排序只需遍历一遍数组，不用执行任何交换操作，时间复杂度为 $O(n)$。

在最坏情况下，冒泡排序要执行 $\frac{(n-1)n}{2}$ 次交换操作，时间复杂度为 $O(n^2)$。

冒泡排序的平均时间复杂度为 $O(n^2)$。

### 2.1.5 代码实现

```
1 void bubble_sort(int a[], int n)
2 {
3     bool flag = true;
4     while (flag)
5     {
```

```
        flag = false;
        for (int i = 0; i < n - 1; i++)
        {
            if (a[i] > a[i + 1])
            {
                flag = true;
                int t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
    return;
}
```

## 2.2 选择排序

### 2.2.1 定义

选择排序（英语：Selection sort）是一种简单直观的排序算法。它的工作原理是每次找出第 $i$ 小的元素（也就是 $A_{i..n}$ 中最小的元素），然后将这个元素与数组第 $i$ 个位置上的元素交换。

### 2.2.2 稳定性

由于 `swap`（交换两个元素）操作的存在，选择排序是一种不稳定的排序算法。

### 2.2.3 时间复杂度

选择排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n^2)$。

### 2.2.4 代码实现

```c
void selection_sort(int a[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int ith = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[ith])
                ith = j;
        int t = a[i];
        a[i] = a[ith];
        a[ith] = t;
    }
    return;
}
```

## 2.3 插入排序

### 2.3.1 定义

插入排序（英语：Insertion sort）是一种简单直观的排序算法。它的工作原理为将待排列元素划分为「已排序」和「未排序」两部分，每次从「未排序的」元素中选择一个插入到「已排序的」元素中的正确位置。

### 2.3.2 稳定性

插入排序是一种稳定的排序算法。

### 2.3.3 时间复杂度

插入排序的最优时间复杂度为 $O(n)$，在数列几乎有序时效率很高。
插入排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$。

### 2.3.4 代码实现

```c
void insertion_sort(int a[], int len)
{
    for (int i = 1; i < len; i++)
    {
        int key = a[i];
        int j = i - 1;
        while (j >= 0 && a[j] > key)
        {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
    return;
}
```

## 2.4 归并排序

### 2.4.1 定义

归并排序（merge sort）是高效的基于比较的稳定排序算法。

### 2.4.2 性质

归并排序基于分治思想将数组分段排序后合并，时间复杂度在最优、最坏与平均情况下均为 $\Theta(n \log n)$，空间复杂度为 $\Theta(n)$。

归并排序可以只使用 $\Theta(1)$ 的辅助空间，但为便捷通常使用与原数组等长的辅助数组。

### 2.4.3 过程

归并排序最核心的部分是合并（merge）过程：将两个有序的数组 `a[i]` 和 `b[j]` 合并为一个有序数组 `c[k]`。

从左往右枚举 a[i] 和 b[j]，找出最小的值并放入数组 c[k]；重复上述过程直到 a[i] 和 b[j] 有一个为空时，将另一个数组剩下的元素放入 c[k]。

为保证排序的稳定性,前段首元素小于或等于后段首元素时（a[i] <= b[j]）而非小于时（a[i] < b[j]）就要作为最小值放入 c[k]。

### 2.4.4 代码实现

```
void merge_sort(int a[], int l, int r)
{
    if (l == r)
        return;
    int mid = (l + r) / 2;
    merge_sort(a, l, mid);
    merge_sort(a, mid + 1, r);
    int k = 0;
    int *b = (int *)malloc(sizeof(int) * (r - l + 1));
    int i = l, j = mid + 1;
    while (i <= mid && j <= r)
    {
        if (a[i] <= a[j])
            b[k] = a[i], i++;
        else
            b[k] = a[j], j++;
        k++;
    }
    while (i <= mid)
        b[k] = a[i], i++, k++;
    while (j <= r)
        b[k] = a[j], j++, k++;
    for (int i = l; i <= r; i++)
        a[i] = b[i - l];
    free(b);
```

```
26      return;
27  }
```

## 2.5 快速排序

### 2.5.1 定义

快速排序（英语：Quicksort），又称分区交换排序（英语：partition-exchange sort），简称「快排」，是一种被广泛运用的排序算法。

### 2.5.2 过程

快速排序的工作原理是通过分治的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系。具体来说，第一步要是要把数列分成两个部分，然后保证前一个子数列中的数都小于后一个子数列中的数。为了保证平均时间复杂度，一般是随机选择一个数 $m$ 来当做两个子数列的分界。

之后，维护一前一后两个指针 $p$ 和 $q$，依次考虑当前的数是否放在了应该放的位置（前还是后）。如果当前的数没放对，比如说如果后面的指针 $q$ 遇到了一个比 $m$ 小的数，那么可以交换 $p$ 和 $q$ 位置上的数，再把 $p$ 向后移一位。当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇。

其实，快速排序没有指定应如何具体实现第一步，不论是选择 $m$ 的过程还是划分的过程，都有不止一种实现方法。

第三步中的序列已经分别有序且第一个序列中的数都小于第二个数，所以直接拼接起来就好了。

### 2.5.3 稳定性

快速排序是一种不稳定的排序算法。

### 2.5.4 时间复杂度

快速排序的最优时间复杂度和平均时间复杂度为 $O(n \log n)$，最坏时间复杂度为 $O(n^2)$。

### 2.5.5 代码实现

```
void quick_sort(int a[], int l, int r)
{
    if (l >= r)
        return;
    int x = a[r];
    int mid = l - 1;
    for (int i = l; i < r; i++)
        if (a[i] <= x)
        {
            mid++;
            int t = a[mid];
            a[mid] = a[i], a[i] = t;
        }
    mid++;
    int t = a[mid];
    a[mid] = a[r], a[r] = t;
    quick_sort(a, l, mid - 1);
    quick_sort(a, mid + 1, r);
    return;
}
```

## 2.6 基数排序

### 2.6.1 定义

基数排序（英语：Radix sort）是一种非比较型的排序算法，最早用于解决卡片排序的问题。基数排序将待排序的元素拆分为 $k$ 个关键字，逐一对各个关键字排序后完成对所有元素的排序。

如果是从第 1 关键字到第 $k$ 关键字顺序进行比较，则该基数排序称为 MSD（Most Significant Digit first）基数排序；

如果是从第 $k$ 关键字到第 1 关键字顺序进行比较，则该基数排序称为 LSD（Least Significant Digit first）基数排序。

### 2.6.2 k - 关键字元素的比较

下面用 $a_i$ 表示元素 $a$ 的第 $i$ 关键字。

假如元素有 $k$ 个关键字，对于两个元素 $a$ 和 $b$，默认的比较方法是：

- 比较两个元素的第 1 关键字 $a_1$ 和 $b_1$，如果 $a_1 < b_1$ 则 $a < b$，如果 $a_1 > b_1$ 则 $a > b$，如果 $a_1 = b_1$ 则进行下一步；
- 比较两个元素的第 2 关键字 $a_2$ 和 $b_2$，如果 $a_2 < b_2$ 则 $a < b$，如果 $a_2 > b_2$ 则 $a > b$，如果 $a_2 = b_2$ 则进行下一步；
- ……
- 比较两个元素的第 $k$ 关键字 $a_k$ 和 $b_k$，如果 $a_k < b_k$ 则 $a < b$，如果 $a_k > b_k$ 则 $a > b$，如果 $a_k = b_k$ 则 $a = b$。

例子：

- 如果对自然数进行比较，将自然数按个位对齐后往高位补齐 0，则一个数字从左往右数第 $i$ 位数就可以作为第 $i$ 关键字；
- 如果对字符串基于字典序进行比较，一个字符串从左往右数第 $i$ 个字符就可以作为第 $i$ 关键字；

### 2.6.3 MSD 基数排序

基于 k - 关键字元素的比较方法，可以想到：先比较所有元素的第 1 关键字，就可以确定出各元素大致的大小关系；然后对具有相同第 1 关键字的元素，再比较它们的第 2 关键字……以此类推。

由于是从第 1 关键字到第 $k$ 关键字顺序进行比较，由上述思想导出的排序算法称为 MSD（Most Significant Digit first）基数排序。

将待排序的元素拆分为 $k$ 个关键字，先对第 1 关键字进行稳定排序，然后对于每组具有相同关键字的元素再对第 2 关键字进行稳定排序（递归执行）……最后对于每组具有相同关键字的元素再对第 $k$ 关键字进行稳定排序。

一般而言，我们默认基数排序是稳定的，所以在 MSD 基数排序中，我们也仅仅考虑借助稳定算法（通常使用计数排序）完成内层对关键字的排序。

正确性参考上文 k - 关键字元素的比较。

### 2.6.4 LSD 基数排序

MSD 基数排序从第 1 关键字到第 $k$ 关键字顺序进行比较，为此需要借助递归或迭代来实现，时间常数还是较大，而且在比较自然数上还是略显不便。

而将递归的操作反过来：从第 $k$ 关键字到第 1 关键字顺序进行比较，就可以得到 LSD（Least Significant Digit first）基数排序，不使用递归就可以完成的排序算法。

将待排序的元素拆分为 $k$ 个关键字，然后先对所有元素的第 $k$ 关键字进行稳定排序，再对所有元素的第 $k-1$ 关键字进行稳定排序，再对所有元素的第 $k-2$ 关键字进行稳定排序……最后对所有元素的第 1 关键字进行稳定排序，这样就完成了对整个待排序序列的稳定排序。

LSD 基数排序也需要借助一种稳定算法完成内层对关键字的排序。同样的，通常使用计数排序来完成。

LSD 基数排序的正确性可以参考《算法导论（第三版）》第 8.3-3 题的解法。

### 2.6.5 代码实现

```
1 void radix_sort(int a[], int n)
2 {
```

```
    int *b = (int *)malloc(sizeof(int) * n);
    int cnt[1 << 8];
    int mask = (1 << 8) - 1;
    int *x = a, *y = b;
    for (int i = 0; i < 32; i += 8)
    {
        for (int j = 0; j != (1 << 8); j++)
            cnt[j] = 0;
        for (int j = 0; j != n; j++)
            ++cnt[x[j] >> i & mask];
        for (int sum = 0, j = 0; j != (1 << 8); j++)
            sum += cnt[j], cnt[j] = sum - cnt[j];
        for (int j = 0; j != n; j++)
            y[cnt[x[j] >> i & mask]++] = x[j];
        int *t = x;
        x = y, y = t;
    }
    free(b);
    return;
}
```

### 2.7 调用标准库函数排序

这里选用了 stdlib.h 库中的 qsort 函数。

#### 2.7.1 代码实现

```
int cmp(const void *a, const void *b)
{
    return (*(int *)a - *(int *)b);
}
void stdlib_sort(int a[], int n)
```

```
{
    qsort(a, n, sizeof(int), cmp);
    return;
}
```

# 3 测试结果

## 3.1 测试环境

本次性能评估在特定的测试环境下进行以确保测试结果的准确性和可靠性。测试环境的详细配置如下：

### 硬件配置

- 处理器：AMD Ryzen 9 7950X
- 内存：64 GB DDR5
- 操作系统：Ubuntu 22.04 LTS，64 位

### 软件配置

- 编程语言：C 语言
- 编译器：GCC version 11.4.0 (Ubuntu 11.4.0-1ubuntu122.04)
- 编译命令：`gcc test.c -o test -Ofast`
- 性能测试工具：C 标准库中的 `clock()` 函数，用于测量程序的执行时间

在此环境下进行的测试旨在提供一个标准化、可控的平台，以便准确地评估和比较不同排序算法的性能。所有测试均在相同的硬件和软件配置下进行，以消除由环境差异导致的测试结果偏差。

为了确保测试的一致性，每种排序算法都被实现为独立的 C 程序，并在相同的编译器和操作系统环境下编译和运行。测试数据集是通过 C 标准库中的随机数生成函数创建，以保证数据的随机性和一致性。在测试过程中，我记录了每个排序算法处理相同数据集时的执行时间，使用的是 C 标准库中的 `clock()` 函数，该命令可以提供程序的执行时间。

### 3.2 测试方法

为了全面评估不同排序算法在 C 语言环境下的性能，本测试采取了以下几个关键步骤：

1. 数据准备：使用 C 标准库中的 `rand()` 函数生成测试数据，采用 142857，114514，1919810 三个随机数种子。为确保算法处理相同数据集，首先生成一个大规模随机数数组，然后将其复制给每个排序算法。数据集包括完全随机、几乎有序和完全逆序三种类型，每种类型分别生成不同规模（$10^3$、$10^4$、$10^5$、$10^6$ 和 $10^7$ 元素）的数据集。

2. 算法实现：所有排序算法均用 C 语言实现。我们重点关注代码优化，确保性能测试能够真实反映算法的潜力。

3. 性能测量：对每种算法和数据集组合使用 `clock()` 函数测量执行时间。每种组合重复多次运行，以获取平均执行时间，减少外部因素引起的误差。

4. 正确性验证：每次排序后，通过检查数组是否升序排列来验证排序结果的正确性，确保性能数据对应于正确执行的算法。

5. 结果记录：记录并整理每种算法在不同数据集上的平均执行时间，以便于后续分析。

完整代码详见附录。

### 3.3 测试结果表格

#### 3.3.1 无序序列

| 运行时间（秒） | $n = 10^3$ | $n = 10^4$ | $n = 10^5$ | $n = 10^6$ | $n = 10^7$ |
|---|---|---|---|---|---|
| 冒泡排序 | 0.001731 | 0.192954 | 21.884712 | 超时 | 超时 |
| 选择排序 | 0.000257 | 0.016128 | 1.648676 | 144.312317 | 超时 |
| 插入排序 | 0.000122 | 0.008000 | 0.729033 | 64.845820 | 超时 |
| 归并排序 | 0.000050 | 0.000565 | 0.006451 | 0.079235 | 0.808949 |
| 快速排序 | 0.000029 | 0.000364 | 0.004611 | 0.055391 | 0.561969 |
| 基数排序 | 0.000008 | 0.000038 | 0.000779 | 0.007510 | 0.063257 |
| `qsort` 函数 | 0.000045 | 0.000584 | 0.007133 | 0.137788 | 0.968817 |

#### 3.3.2 几乎有序序列

| 运行时间（秒） | $n = 10^3$ | $n = 10^4$ | $n = 10^5$ | $n = 10^6$ | $n = 10^7$ |
|---|---|---|---|---|---|
| 冒泡排序 | 0.000731 | 0.033247 | 5.431512 | 超时 | 超时 |
| 选择排序 | 0.000236 | 0.014007 | 1.643990 | 144.180923 | 超时 |
| 插入排序 | 0.000007 | 0.000021 | 0.000271 | 0.003165 | 0.002110 |
| 归并排序 | 0.000023 | 0.000179 | 0.001969 | 0.018667 | 0.243730 |
| 快速排序 | 0.000135 | 0.005410 | 0.976079 | 83.249098 | 超时 |
| 基数排序 | 0.000022 | 0.000186 | 0.003059 | 0.006309 | 0.058694 |
| `qsort` 函数 | 0.000008 | 0.000041 | 0.000667 | 0.030576 | 0.322435 |

**Remark 1** 超时是指运行时间超过 600 秒。

### 3.4 结果分析

#### 3.4.1 理论时间复杂度分析

- 冒泡排序、简单选择排序、简单插入排序通常都具有 $O(n^2)$ 的时间复杂度，在数据规模较大时性能不佳。
- 归并排序和快速排序的平均时间复杂度为 $O(n \log n)$，在各种规模数据上通常表现良好，但快速排序在最坏情况下会退化为 $O(n^2)$。
- 基数排序不基于比较，时间复杂度为 $O(n)$，在所有数据集下有优异的性能表现。
- 标准库排序函数通常实现了高效的排序算法，旨在处理各种数据场景，通常具有很好的平均性能。

#### 3.4.2 实际测试结果分析

- 在不同规模的模拟数据上，各排序算法的性能差异显著。
- 对于小规模数据，简单排序算法（如冒泡、选择、插入排序）可能表现出与更复杂算法相当的性能，但随着数据规模增大，它们的性能逐渐劣化。
- 归并排序和快速排序在大多数情况下表现良好，尤其是快速排序在随机数据上通常比归并排序更快，但在最坏情况（如已经排序的数组）下可能会表现不佳。
- 在大规模数据下，冒泡、选择、插入排序的时间差距较大。可能是因为冒泡排序的实现通常包含更多的交换操作，每次交换涉及多个赋值操作，这在实践中是代价较高的。选择排序减少了交换次数，但比较次数依然较高。由于交换通常比比较更耗时，这使得选择排序在某些情况下比冒泡排序更有效。插入排序在部分有序的数组中表现最佳。由于它只在必要时才移动元素，因此可以减少不必要的交换操作。
- 基数算法在所有条件下表现出色。
- 标准库函数排序在所有测试中表现中规中矩。

### 3.4.3 几乎有序数组的排序性能分析

- 在几乎有序的数组中，简单插入排序可能显示出意外的高效率，因为它在数据已经部分排序的情况下性能最佳。
- 归并排序和快速排序在几乎有序的数据上可能不会显示出显著的性能改善，尤其是快速排序，可能因为递归深度增加而表现较差。
- 标准库排序函数考虑到了各种可能的数据排列，因此即使在几乎有序的数据上也能维持高效的排序性能。

### 3.4.4 超时和性能瓶颈的分析

- 在极大规模数据上，$O(n^2)$ 的算法几乎总是不可行的，这在测试结果中应有所体现。
- 对于特定算法的性能瓶颈，如快速排序的最坏情况，分析其出现的条件和如何通过算法优化（例如引入随机化）来改善。

# 4 小结

在本报告中，我们对多种排序算法进行了详细的分析和性能评测，包括冒泡排序、选择排序、插入排序、归并排序、快速排序，以及至少一种特殊排序算法。这些算法在不同的数据规模（$10^3$、$10^4$、$10^5$、$10^6$、$10^7$）和不同的数据状态（完全无序和几乎有序）下进行了测试。

- 基本排序算法（冒泡、选择、插入）在小规模数据集上相对有效，但在大规模数据处理上表现不佳，主要由于它们的 $O(n^2)$ 时间复杂度。
- 归并排序和快速排序在大多数情况下表现出色，特别是在大规模数据集上，它们的 $O(n \log n)$ 时间复杂度提供了显著的性能优势，但快速排序在最坏情况下会退化为 $O(n^2)$。
- 基数排序在特定条件下展现出优异的性能，特别是数据值域较小的时候。
- 标准库排序函数表现出中规中矩的 $O(n \log n)$ 排序算法的综合性能，适应了各种数据规模和数据状态。

在几乎有序的数据集上，简单插入排序显示出其高效性，特别是当数据集已经部分排序时。此外，我们还观察到，不同排序算法在理论复杂度和实际性能之间存在差异，这可能归因于数据的特性和算法实现的常数因子。

# A 附录

## A.1 完整测试代码

sort.h

```
#include <stdlib.h>
#include <stdbool.h>

void bubble_sort(int a[], int n)
{
    bool flag = true;
    while (flag)
    {
        flag = false;
        for (int i = 0; i < n - 1; i++)
        {
            if (a[i] > a[i + 1])
            {
                flag = true;
                int t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
    return;
}

void selection_sort(int a[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
```

```
28          int ith = i;
29          for (int j = i + 1; j < n; j++)
30              if (a[j] < a[ith])
31                  ith = j;
32          int t = a[i];
33          a[i] = a[ith];
34          a[ith] = t;
35      }
36      return;
37  }
38
39  void insertion_sort(int a[], int len)
40  {
41      for (int i = 1; i < len; i++)
42      {
43          int key = a[i];
44          int j = i - 1;
45          while (j >= 0 && a[j] > key)
46          {
47              a[j + 1] = a[j];
48              j--;
49          }
50          a[j + 1] = key;
51      }
52      return;
53  }
54
55  void merge_sort_runner(int a[], int l, int r)
56  {
57      if (l == r)
58          return;
```

```
59      int mid = (l + r) / 2;
60      merge_sort_runner(a, l, mid);
61      merge_sort_runner(a, mid + 1, r);
62      int k = 0;
63      int *b = (int *)malloc(sizeof(int) * (r - l + 1));
64      int i = l, j = mid + 1;
65      while (i <= mid && j <= r)
66      {
67          if (a[i] <= a[j])
68              b[k] = a[i], i++;
69          else
70              b[k] = a[j], j++;
71          k++;
72      }
73      while (i <= mid)
74          b[k] = a[i], i++, k++;
75      while (j <= r)
76          b[k] = a[j], j++, k++;
77      for (int i = l; i <= r; i++)
78          a[i] = b[i - l];
79      free(b);
80      return;
81  }
82  void merge_sort(int a[], int n)
83  {
84      return merge_sort_runner(a, 0, n - 1);
85  }
86
87  void quick_sort_runner(int a[], int l, int r)
88  {
89      if (l >= r)
```

```
90          return;
91      int x = a[r];
92      int mid = l - 1;
93      for (int i = l; i < r; i++)
94          if (a[i] <= x)
95          {
96              mid++;
97              int t = a[mid];
98              a[mid] = a[i], a[i] = t;
99          }
100     mid++;
101     int t = a[mid];
102     a[mid] = a[r], a[r] = t;
103     quick_sort_runner(a, l, mid - 1);
104     quick_sort_runner(a, mid + 1, r);
105     return;
106 }
107
108 void quick_sort(int a[], int n)
109 {
110     return quick_sort_runner(a, 0, n - 1);
111 }
112
113 int cmp(const void *a, const void *b)
114 {
115     return (*(int *)a - *(int *)b);
116 }
117 void stdlib_sort(int a[], int n)
118 {
119     qsort(a, n, sizeof(int), cmp);
120     return;
```

```
}

void radix_sort(int a[], int n)
{
    int *b = (int *)malloc(sizeof(int) * n);
    int cnt[1 << 8];
    int mask = (1 << 8) - 1;
    int *x = a, *y = b;
    for (int i = 0; i < 32; i += 8)
    {
        for (int j = 0; j != (1 << 8); j++)
            cnt[j] = 0;
        for (int j = 0; j != n; j++)
            ++cnt[x[j] >> i & mask];
        for (int sum = 0, j = 0; j != (1 << 8); j++)
            sum += cnt[j], cnt[j] = sum - cnt[j];
        for (int j = 0; j != n; j++)
            y[cnt[x[j] >> i & mask]++] = x[j];
        int *t = x;
        x = y, y = t;
    }
    free(b);
    return;
}
```

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include ”sort.h”

```

```
bool is_sorted(int a[], int n)
{
    for (int i = 1; i < n; i++)
        if (a[i] < a[i - 1])
            return false;
    return true;
}

void generate(int *a, int *b, int n)
{
    for (int i = 0; i < n; i++)
        a[i] = b[i] = rand();
    stdlib_sort(b, n);
    int times = 10;
    for (int i = 1; i <= times; i++)
    {
        int l = rand() % n, r = rand() % n;
        while (l == r)
            l = rand() % n, r = rand() % n;
        int t = b[l];
        b[l] = b[r];
        b[r] = t;
    }
    return;
}

void test(int a[], int n, void (*sort)(int *, int), const
   char type[], const char name[])
{
    int *c = (int *)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++)
```

```
37          c[i] = a[i];
38      int st = clock();
39      sort(c, n);
40      int ed = clock();
41      double time = (double)(ed - st) / CLOCKS_PER_SEC;
42      printf("%20s | %25s | time: %.10lf\n", type, name,
            time);
43      if (!is_sorted(c, n))
44      {
45          fprintf(stderr, "Error\n");
46          exit(1);
47      }
48      free(c);
49      return;
50  }
51
52  int main(int argc, char *argv[])
53  {
54      if (argc < 3)
55      {
56          fprintf(stderr, "Need more parameters");
57          return 1;
58      }
59      int n = atoi(argv[1]), seed = atoi(argv[2]);
60      printf("n = %d, seed = %d\n", n, seed);
61      srand(seed);
62      int *a = (int *)malloc(sizeof(int) * n), *b = (int *)
            malloc(sizeof(int) * n);
63      generate(a, b, n);
64      test(a, n, bubble_sort, "Bubble Sort", "Random
            Sequence");
```

```
65      test(b, n, bubble_sort, "Bubble Sort", "Almost Ordered
            Sequence");
66      test(a, n, selection_sort, "Selection Sort", "Random
            Sequence");
67      test(b, n, selection_sort, "Selection Sort", "Almost
            Ordered Sequence");
68      test(a, n, insertion_sort, "Insertion Sort", "Random
            Sequence");
69      test(b, n, insertion_sort, "Insertion Sort", "Almost
            Ordered Sequence");
70      test(a, n, merge_sort, "Merge Sort", "Random Sequence"
            );
71      test(b, n, merge_sort, "Merge Sort", "Almost Ordered
            Sequence");
72      test(a, n, quick_sort, "Quick Sort", "Random Sequence"
            );
73      test(b, n, quick_sort, "Quick Sort", "Almost Ordered
            Sequence");
74      test(a, n, radix_sort, "Radix Sort", "Random Sequence"
            );
75      test(b, n, radix_sort, "Radix Sort", "Almost Ordered
            Sequence");
76      test(a, n, stdlib_sort, "Stdlib Sort", "Random
            Sequence");
77      test(b, n, stdlib_sort, "Stdlib Sort", "Almost Ordered
            Sequence");
78      return 0;
79  }
```