算法基础 实验一

算法基础 实验一

- 1 实验环境
- 2 实验内容
- 3 实验方法与步骤
  - 3.1 编写排序函数
  - 3.2 编写时间测量模块
  - 3.3 编写主程序
  - 3.4 编译优化
- 4 实验结果
  - 4.1 程序输出
  - 4.2 结果分析
  - 4.3 与 `std::sort` 进行对比

1 实验环境

本次实验所使用的环境为 `Linux(5.4.65-1-pve)/Ubuntu(4.15.99)/x86_64`

编译器采用 `g++-8` 。

SIMD 指令集采用 `SSE2` 。

由于本实验代码使用到 c++17 `std::filesystem` 。所以请务必使用 `g++-8` 进行编译。对于 Ubuntu，请使用 `apt install g++-8` 安装。具体编译方法可以参照 `Makefile` 。

2 实验内容

- **排序 n 个元素，元素为随机生成的 0 到 $2^{15}-1$ 之间的整数，n 的取值为：$2^3, 2^6, 2^9\ 2^{12}, 2^{15}, 2^{18}$。**

  本次实验中，由于代码运行速度足够快，所以 n 就简单取值为 $2^3, 2^4, 2^5 \ldots, 2^{18}$，这样可以得到更为连续的曲线。

  另外，本次实验使用 `uint16_t` 进行排序，满足 0 到 $2^{15}-1$ 之间的整数的要求，并尽可能给编译器优化的空间。

- **实现以下算法：插入排序，堆排序，快速排序，归并排序，计数排序。**

  本次实现实现了这五种排序，并与c++已经实现的 `std::sort` 进行对比。

3 实验方法与步骤

3.1 编写排序函数

为了使我们的排序函数方便使用，我将自己编写的几个排序函数的使用方法与 C++ STL 的 `std::sort` 保持一致，下面以插入排序为例来看一下函数的使用方法。

```
namespace insertion_sort {

    /**
     * @brief Insertion sort at a Random iterator.
     *
     * @tparam RandomIt A type of LegacyRandomAccessIterator.
     * @param begin
     * @param end
     */
    template<typename RandomIt>
    void insertion_sort(RandomIt begin, RandomIt end) {
        for(auto q = begin + 1; q != end; q++) {
            auto key = *q;
            auto p = q - 1;
            while(p >= begin && *p > key) {
                *(p+1) = *p;
                --p;
            }
            *(p+1) = key;
        }
    }

} // namespace insertion_sort
```

使用的时候之间调用即可：

```
insertion_sort::insertion_sort(vec.begin(), vec.end());
```

然后按照课本算法，分别编写插入排序、归并排序，堆排序、快速排序、计数排序，这些排序函数的声明如下：

```
template<typename RandomIt>
void insertion_sort::insertion_sort(RandomIt begin, RandomIt end);

template<typename RandomIt>
void heap_sort::heap_sort(RandomIt begin, RandomIt end);

template<typename RandomIt>
void merge_sort::merge_sort(RandomIt begin, RandomIt end);

template<typename RandomIt>
void quick_sort::quick_sort(RandomIt begin, RandomIt end);

template<typename RandomIt>
void count_sort::count_sort(RandomIt begin, RandomIt end);
```

3.2 编写时间测量模块

这个计时模块（ `simple_metrics` ）是我从一篇论文所使用的代码摘下来的。这里使用 C++ 的 RAII 机制来简化复杂的计时过程。使用 `std::chrono::high_resolution_clock` 来获取精准的纳秒级的时间，并写入到响应到文件。

```
// 使用 RAII 机制计时。
class RaiiTimer {
    namespace fs = std::filesystem;
    namespace chrono = std::chrono;

    using high_resolution_clock = chrono::high_resolution_clock;
    using time_point = chrono::time_point<std::chrono::high_resolution_clock>;

    std::ofstream file;
    time_point start_time;
    public:

    RaiiTimer(const fs::path& path)
        : file(path, std::ios_base::app) /* 以追加格式打开文件*/ {
        start_time = high_resolution_clock::now();
    }

    ~RaiiTimer() {
        auto end_time = high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time -
 start_time).count();
        file << duration << std::endl;
    }
};


```

使用方法如下：

```
#define METRICS_FUNC_TIME(label) auto&& __timer =
simple_metrics::createRaiiTimer(label);

void func_needs_count_time() {
    METRICS_FUNC_TIME("func_needs_count_time");
    // time-cost procedure
}
```

这样就可以在该函数开始时自动计时，在结束时自动保存花费的时间。

3.3 编写主程序

利用 C++ 强大的抽象能力，我使用一个函数来表示一次测试，该函数包含一次测试所需的所有信息，该函数声明如下：

```
/**
```

```
 * @brief Sort the vector and save the result to <output_path>/<sort_method_name>/<result_file_name>.
 *
 * @tparam F sort function type for sort_func.
 * @tparam T sort element type. (e.g. int, uint32_t)
 * @param sort_func sort function (e.g. count_sort::count_sort<IteratorType>, std::sort<IteratorType>)
 * @param copied_vec input vector (copied to avoid changing the original input).
 * @param size size of input
 * @param output_path path of the folder 'output/'
 * @param sort_method_name name of the sort method (e.g. count_sort)
 * @param result_file_name name of the result file (e.g. result_n.txt)
 */
template<typename F, typename T>
inline void sort_and_save(
    F sort_func,
    std::vector<T> input_vec,
    std::size_t size,
    const fs::path& output_path,
    const std::string& sort_method_name,
    const std::string& result_file_name
);
```

该函数会调用 sort_func 来运行排序算法，使用 `METRICS_FUNC_TIME` 来计时，并将结果保存到 `<output_path>/<sort_method_name>/<result_file_name>` ，并输出简单的调试信息。

该函数的使用方法：

```
// 获取迭代器的类型
using IterT = std::vector<uint16_t>::iterator;

// 获取result_n.txt文件的文件名
auto result_file_name = "result_"s + std::to_string(i) + ".txt";

// 调用 sort_and_save，输入排序函数 heap_sort::heap_sort<IterT>、向量和需要排序的部分的大小、输出的路径。
// 该函数表示一次完整的测试过程。
sort_and_save(heap_sort::heap_sort<IterT>, input_vec, 1 << i,
              output_path, "heap_sort", result_file_name);
```

然后反复调用上面的函数，就可以得到我们需要的结果。

3.4 编译优化

以上虽然是使用C++的 `std::vector` 编写，但是它可以比某些同学使用 C 写的还要更快（插入排序 2^{18} 仅使用 10 秒）。要点就在于要把编译器的优化选项打开（使用 `O2` 就足够了，其他优化也可以用），可以把几乎所有的运算时间都缩小一半。这样就可以几乎不怎么花时间地把 $2^3, 2^4, 2^5 \ldots, 2^{18}$ 规模的输入全部测完，得到更为连续的曲线。

4 实验结果

4.1 程序输出

实验输出：

```
dnailz@VM1765-Dnailz:~/cpptest/algolab1/ex1/src$ make -B run
g++-8 main.cpp simple_metrics/metrics.cpp -o algo_lab1 -std=c++17 -O2 -march=native -I. -lstdc++fs
./algo_lab1 ../input ../output
input_size = 8
method = heap_sort
        time cost(ns): 1130
        sort result = 4976 10709 17951 21628 21854 23728 27294 29786
method = insertion_sort
        time cost(ns): 437
        sort result = 4976 10709 17951 21628 21854 23728 27294 29786
method = merge_sort
        time cost(ns): 1657
        sort result = 4976 10709 17951 21628 21854 23728 27294 29786
method = quick_sort
        time cost(ns): 528
        sort result = 4976 10709 17951 21628 21854 23728 27294 29786
method = count_sort
        time cost(ns): 250995
        sort result = 4976 10709 17951 21628 21854 23728 27294 29786
method = std::sort
        time cost(ns): 882
        sort result = 4976 10709 17951 21628 21854 23728 27294 29786
input_size = 16
method = heap_sort
        time cost(ns): 1193
method = insertion_sort
        time cost(ns): 618
method = merge_sort
        time cost(ns): 3214
method = quick_sort
        time cost(ns): 814
method = count_sort
        time cost(ns): 97859
method = std::sort
        time cost(ns): 741
input_size = 32
method = heap_sort
        time cost(ns): 2368
method = insertion_sort
        time cost(ns): 890
method = merge_sort
        time cost(ns): 5914
method = quick_sort
        time cost(ns): 1387
method = count_sort
        time cost(ns): 95899
method = std::sort
        time cost(ns): 1502
input_size = 64
method = heap_sort
        time cost(ns): 4961
method = insertion_sort
        time cost(ns): 1800
```

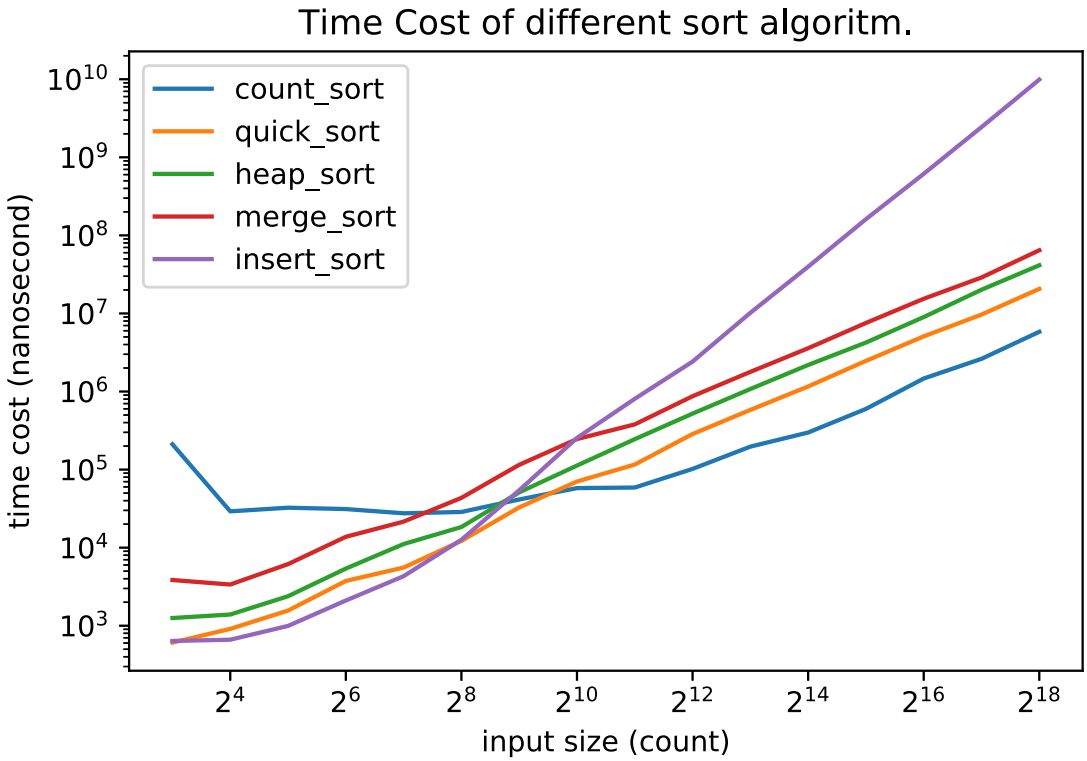
单个排序算法所花费的时间保存在 `time.txt` 中：

ex1 > output > insertion_sort > time.txt

```
16    437
15    618
14    890
13    1800
12    4033
11    14775
10    48935
 9    154538
 8    653828
 7    2603446
 6    10162391
 5    38422586
 4    166424031
 3    639757896
 2    2508659274
 1    10061940294
17
```

4.2 结果分析

将运行的结果绘制成折线图，得到：



程序第一次测量时测量的是 2^3 规模的时间，由于系统原因，该次时间由于系统还没有准备好相应的内存空间，需要进行 mmap 调用所导致的，这在 merge_sort 和 count_sort 中非常明显。故该次数据舍去不用。

1. **分析各个算法的时间复杂度：**

   对运行时间的 2 对数和数据规模的 2 对数进行线性拟合，得到如下的结果：

$$
\begin{aligned}
\text{count\_sort:} &\quad \log_2 T = 0.4366N + 12.6 \\
\text{quick\_sort:} &\quad \log_2 T = 1.035N + 5.623 \\
\text{heap\_sort:} &\quad \log_2 T = 1.053N + 6.261 \\
\text{merge\_sort:} &\quad \log_2 T = 0.9915N + 7.885 \\
\text{insertion\_sort:} &\quad \log_2 T = 1.803N + 1.447
\end{aligned}
$$

   可以看到，insertion_sort 的斜率接近 2，故可以认为其时间复杂度接近于 $O(n^2)$，斜率略小于2是由于在规模小于 2^{10} 的时候，系统的运行时间受比n^2 更小的项影响，会比拟合估计的运行时间更大，进而使得斜率变小。

   quick_sort，heap_sort，merge_sort 的时间复杂度应当 $T = O(n \lg n)$，故 $\lg T = \lg n + \lg \lg n$ 由于 $\lg \lg n$ 很小，可以忽略不计，故这里的斜率接近于 1。

   count_sort 在规模大于 2^{12} 接近于线性，但在小于 2^{12} 的时候，仍然要分配为$O(k)$ 的空间且具有 $O(k)$ 的时间复杂度，故其时间复杂度接近于 $O(n + k)$。

2. **在不同情况下选择不同的算法**

   在输入规模小于2^7 时，应当选择插入排序，插入排序算法简单，适用于小规模的数据。

   在输入规模大于2^7 时，快速排序比堆排序和归并排序有更小的常数，故当优先使用快速排序。

   在一些特殊情况，如输入数据范围很小时（$n > k$），应当采用计数排序。

4.3 与 `std::sort` 进行对比

为了证明我们的排序算法编写得足够地高效，增强以上结论的普适性，需要将本次实验中实现的算法与业界常用的实现进行比较。我们将我们实现的 `quick_sort` 与 C++ STL 提供的 `std::sort` 进行比较，发现我们的实现效率上与 `std::sort` 非常接近。这可以说明我们的算法实现与业内先进水平相接近。