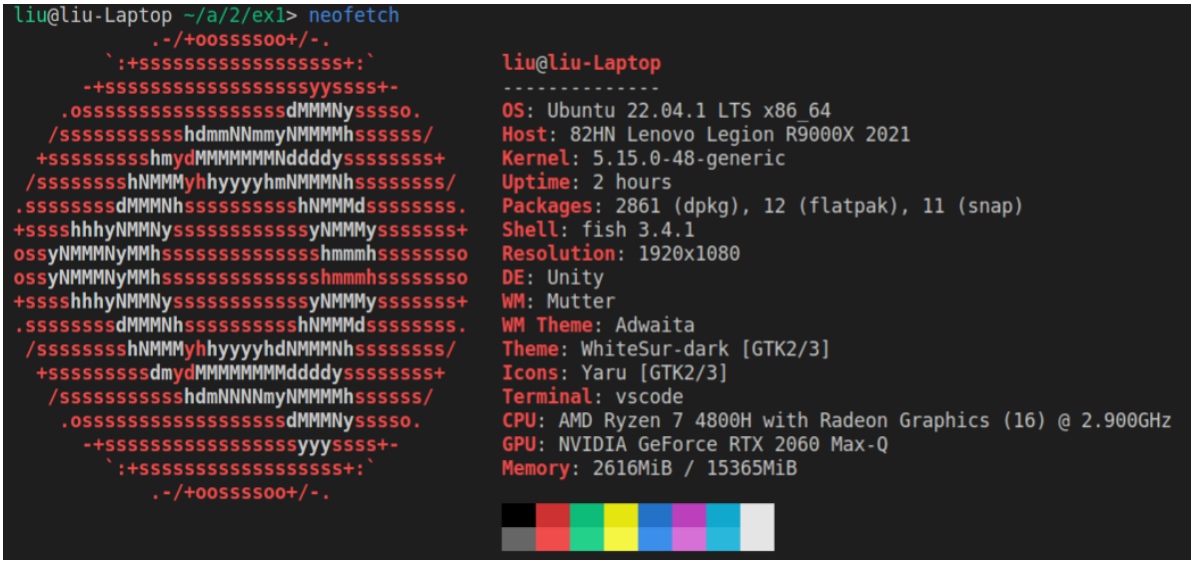
# 算法基础 排序算法实验报告

PB20000180 刘良宇

## 实验内容

比较计数排序，堆排序，归并排序，快速排序在不同数据规模时的排序用时

## 实验设备和环境




注：本实验代码引入了 `unistd.h` 用于解析命令行参数，因此只能在 `POSIX` 系统下编译

## 实验方法和步骤

### 基本思路

分别编写四个排序函数和命令行程序，通过参数指定测试的函数及数据规模

### 计数排序

```cpp
#include <array>
#include <vector>

inline void counting_sort(std::vector<int> &A) {
    constexpr int value_max = 1 << 15;
    std::array<int, value_max> cnt{};
    auto n = A.size();
    auto B = std::vector<int>(n);
    // count
    for (auto i = 0; i < n; i++)
        cnt[A[i]]++;
    // prefix sum
```

```
    for (auto i = 1; i < value_max; i++)
        cnt[i] += cnt[i - 1];
    // sort
    for (auto i = n; i-- > 0;)
        B[--cnt[A[i]]] = A[i];
    // copy
    for (auto i = 0; i < n; i++)
        A[i] = B[i];
}
```

四次循环完成排序。限制值域 $2^{15}$

## 堆排序

堆排序，归并排序和快速排序都是基于比较的排序，所以下面都使用 `Vector<T>` 的模板作为函数输入

```
#include <vector>

template <typename T> void heap_sort(std::vector<T> &A) {
    auto heap_size = A.size();
    auto max_heapify = [&](std::size_t index) {
        while (true) {
            auto l = index * 2 + 1, r = index * 2 + 2;
            auto largest = index;
            if (l < heap_size && A[l] > A[index]) {
                largest = l;
            }
            if (r < heap_size && A[r] > A[largest]) {
                largest = r;
            }
            if (largest == index) {
                return;
            }
            std::swap(A[index], A[largest]);
            index = largest;
        }
    };
    // build max heap
    for (auto i = A.size() / 2; i-- > 0;) {
        max_heapify(i);
    }
    // sort by poping the biggest and continue
    for (auto i = A.size() - 1; i > 0; i--) {
        std::swap(A[0], A[i]);
        heap_size -= 1;
        max_heapify(0);
    }
}
```

基本思路与书上伪代码相同，先建立大根堆，再逐个选出最大值，完成排序

## 归并排序

```cpp
#include <vector>

template <typename T> void merge_sort(std::vector<T> &A) {
    // temp vector for merge, B.size() ==  A.size()
    std::vector<T> B(A.size());
    auto _merge_sort = [&](auto &&self, std::size_t lo, std::size_t hi) {
        if (lo >= hi)
            return;
        auto mid = (lo + hi) / 2 + 1;

        // divide into 2 subproblems
        self(self, lo, mid - 1);
        self(self, mid, hi);

        // merge with temp vector B
        auto left = lo, right = mid;
        for (auto ans = lo; ans <= hi; ans++) {
            if (left >= mid || (right <= hi && A[left] > A[right])) {
                B[ans] = A[right++];
            } else {
                B[ans] = A[left++];
            }
        }
        // copy B into A
        for (auto i = lo; i <= hi; i++)
            A[i] = B[i];
    };
    _merge_sort(_merge_sort, 0, A.size() - 1);
}
```

这里在函数内部新建一个 `Vector<T> B` 作为归并时的公用临时数组

因为这里实现的是一般的基于比较的排序，所以不使用哨兵

## 快速排序

```cpp
#include <vector>

template <typename T>
inline std::size_t partition(std::vector<T> &A, std::size_t lo,
                             std::size_t hi) {
    T pivot = A[(lo + hi) / 2];
    while (true) {
        while (A[lo] < pivot)
            lo++;
        while (A[hi] > pivot)
            hi--;
        if (lo >= hi)
            return hi;
        std::swap(A[lo], A[hi]);
        lo++, hi--;
    }
}

template <typename T>
```

```cpp
void quick_sort_(std::vector<T> &A, std::size_t lo, std::size_t hi) {
    if (lo >= hi || lo < 0)
        return;
    auto pivot = partition(A, lo, hi);
    quick_sort_(A, lo, pivot); // pivot is not the actual pivot position
    quick_sort_(A, pivot + 1, hi);
    return;
}

// for main.cpp
template <typename T> void quick_sort(std::vector<T> &A) {
    quick_sort_(A, 0, A.size() - 1);
}
```

采用哨兵以尽可能减少 `partition` 需要的比较次数

## mian

程序的功能分为以下几部分，都通过命令行参数指定：

- 是否随机生成数据，写入 `input/input.txt`
- 指定需要测试的数据规模 $n$（可指定多个）
- 指定需要测试的排序方法 $m$（可指定多个）
- 保存结果到对应 `result_n.txt` 和 `time.txt` 并打印

### 随机生成数据

```cpp
// generate 2 ** 18 unsigned in [0,  2 ** 15 - 1]
srand((unsigned)time(nullptr));
if (generate_number) {
    auto gen_rand_file = std::ofstream(file_path);
    for (auto i = 0; i < (1 << 18); i++) {
        auto gen_number = rand() & ((1 << 15) - 1);
        gen_rand_file << gen_number << std::endl;
    }
    gen_rand_file.close();
}
```

### 指定规模

存储在 `std::set` 里即可

```cpp
std::set<unsigned> testing_scales{};
```

### 指定测试排序方法

```cpp
// sorting methods supplied and tested
std::map<std::string, std::function<void(std::vector<int> &)>> methods_map =
    {{"heap", heap_sort<int>},
     {"quick", quick_sort<int>},
     {"merge", merge_sort<int>},
     {"counting", counting_sort}};
std::set<std::string> sorting_methods{"heap", "quick", "merge", "counting"};
std::set<std::string> testing_methods{};
```

使用 `std::map` 建立参数到实际测试函数的映射

### 测试流程

```
// 读入完整的测试文件的数据到这个数组
auto nums = std::vector<int>(1 << 18);
......

// 测试每一种情况
struct timeval t1, t2;
double timeuse_ms;
for (const auto &scale : testing_scales) {
    for (const auto &method : testing_methods) {
        // 截取完整的数组，得到要测试的数组
        auto test_vector =
            std::vector<int>(nums.begin(), nums.begin() + (1 << scale));
        auto test_function = methods_map[method];

        gettimeofday(&t1, nullptr);
        test_function(test_vector);
        gettimeofday(&t2, nullptr);

        timeuse_ms = (double)(t2.tv_sec - t1.tv_sec) * 1000.0 +
                     (double)(t2.tv_usec - t1.tv_usec) / 1000.0;
        // 这里输出用时，略
        ......

        // 这里保存到文件，略
        ......
    }
}
```

遍历每一种需要测试的情况后计时，排序，处理输出即可

## Makefile

可以一键 `make test` 和 `make clean`

```
.PHONY: all

all: build/main


build/main: src/main.cpp
    g++ -O2 src/main.cpp -o build/main

test: all
    ./build/main -i input/input.txt -g -n3 -n6 -n9 -n12 -n15 -n18 -mheap -mquick -
mcounting -mmerge
    diff -r -xtime.txt output/counting_sort/ output/heap_sort/
    diff -r -xtime.txt output/counting_sort/ output/merge_sort/
    diff -r -xtime.txt output/counting_sort/ output/quick_sort/

clean:
    rm -f build/main
    rm -f output/counting_sort/*.txt output/quick_sort/*.txt output/merge_sort/*.txt
output/heap_sort/*.txt
```

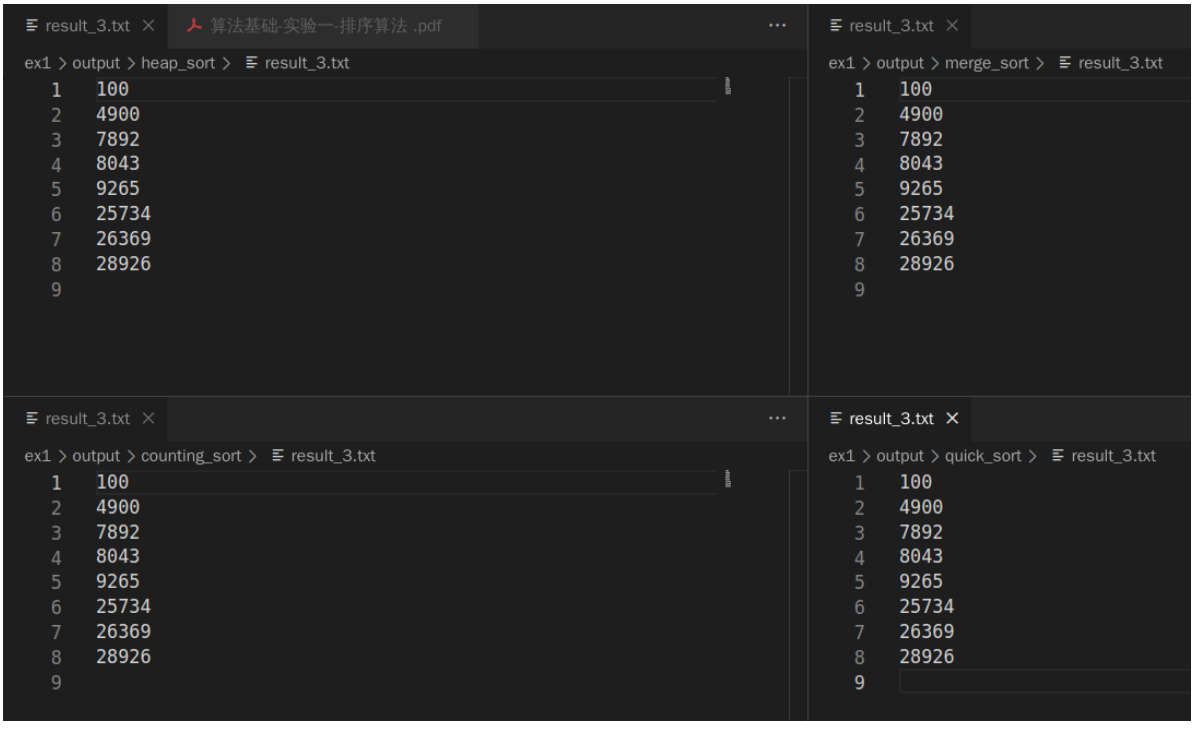
- `make test` 会重新生成随机数输入，指定测试所有的数据规模和所有的排序方法

    - 并且会 `diff` 输出文件夹以辅助检查排序算法的正确性
- `make clean` 清理产生的临时文件

# 实验结果与分析

```
liu@liu-Laptop ~/a/2/ex1> make test
g++ -O2 src/main.cpp -o build/main
./build/main -i input/input.txt -n3 -n6 -n9 -n12 -n15 -n18 -mheap -mquick -mcounting -mmerge
Scale: 3 Method: counting sort time usage: 0.231ms
Scale: 3 Method: heap sort time usage: 0.002ms
Scale: 3 Method: merge sort time usage: 0.003ms
Scale: 3 Method: quick sort time usage: 0.001ms
Scale: 6 Method: counting sort time usage: 0.198ms
Scale: 6 Method: heap sort time usage: 0.006ms
Scale: 6 Method: merge sort time usage: 0.012ms
Scale: 6 Method: quick sort time usage: 0.006ms
Scale: 9 Method: counting sort time usage: 0.154ms
Scale: 9 Method: heap sort time usage: 0.035ms
Scale: 9 Method: merge sort time usage: 0.056ms
Scale: 9 Method: quick sort time usage: 0.049ms
Scale: 12 Method: counting sort time usage: 0.195ms
Scale: 12 Method: heap sort time usage: 0.142ms
Scale: 12 Method: merge sort time usage: 0.223ms
Scale: 12 Method: quick sort time usage: 0.192ms
Scale: 15 Method: counting sort time usage: 0.231ms
Scale: 15 Method: heap sort time usage: 1.445ms
Scale: 15 Method: merge sort time usage: 2.122ms
Scale: 15 Method: quick sort time usage: 1.774ms
Scale: 18 Method: counting sort time usage: 1.258ms
Scale: 18 Method: heap sort time usage: 16.172ms
Scale: 18 Method: merge sort time usage: 19.876ms
Scale: 18 Method: quick sort time usage: 14.595ms
diff -r -xtime.txt output/counting_sort/ output/heap_sort/
diff -r -xtime.txt output/counting_sort/ output/merge_sort/
diff -r -xtime.txt output/counting_sort/ output/quick_sort/
```

查看排序结果：




可以指定生成单个排序方法的结果：

```
liu@liu-Laptop ~/a/2/ex1> ./build/main -i input/input.txt -n3 -n6 -n9 -n12 -n15 -n18 -mheap
Scale: 3 Method: heap sort time usage: 0.001ms
Scale: 6 Method: heap sort time usage: 0.004ms
Scale: 9 Method: heap sort time usage: 0.016ms
Scale: 12 Method: heap sort time usage: 0.14ms
Scale: 15 Method: heap sort time usage: 1.425ms
Scale: 18 Method: heap sort time usage: 17.357ms
```

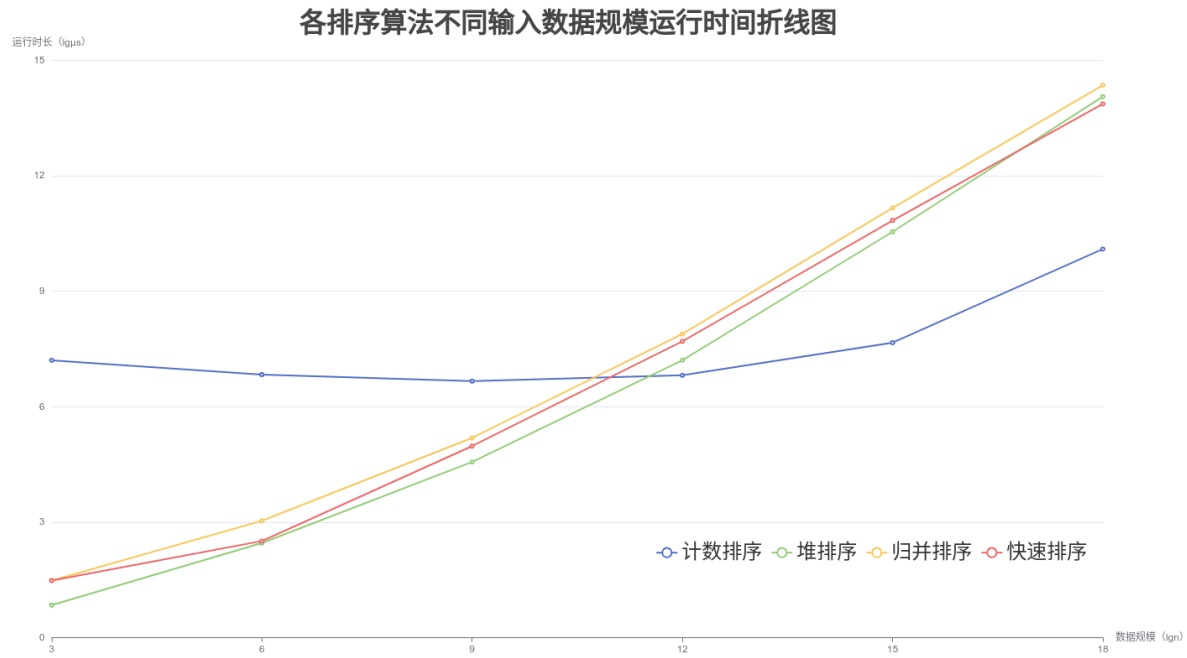
## 数据处理

我们重复运行 10 次 `make test`，结果每次追加保存在了对应的 `time.txt` 中

可以对原始数据进行处理，取平均值，得到下列表格（单位：ms）

| lgn | 3 | 6 | 9 | 12 | 15 | 18 |
|---|---|---|---|---|---|---|
| 计数排序 | 0.1477 | 0.1142 | 0.1015 | 0.1127 | 0.2029 | 1.0938 |
| 堆排序 | 0.0018 | 0.0055 | 0.0237 | 0.1479 | 1.4946 | 17.0361 |
| 归并排序 | 0.0028 | 0.0082 | 0.0365 | 0.2369 | 2.2962 | 20.9264 |
| 快速排序 | 0.0028 | 0.0057 | 0.0315 | 0.2080 | 1.8323 | 14.9326 |

下面根据表格数据绘图：



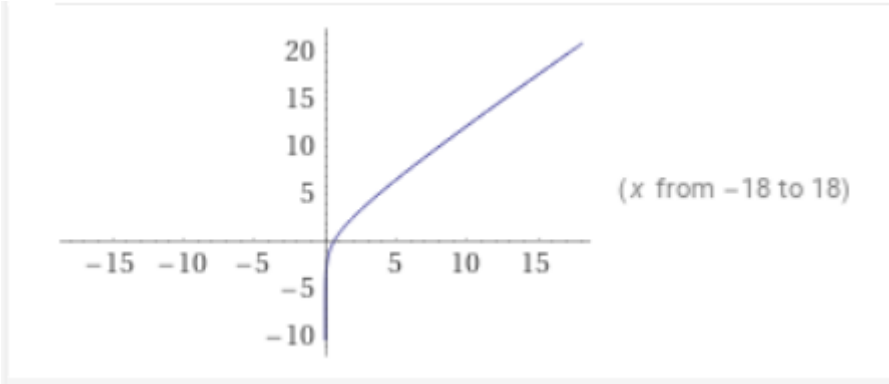

为了方便观看：

- 横坐标为 $\lg n$，即数据规模
- 纵坐标调整为 $lg(\text{运行时间})$，单位：微秒

## 复杂度分析

其中，堆排序，归并排序，快速排序理论时间复杂度 $\Theta(n\lg n)$，$x=\lg n$，则复杂度 $\Theta(xe^x)$

取对数后，理论上纵轴关于横轴的关系式 $y=\lg(xe^x)=x+\lg x$，几乎是线性的

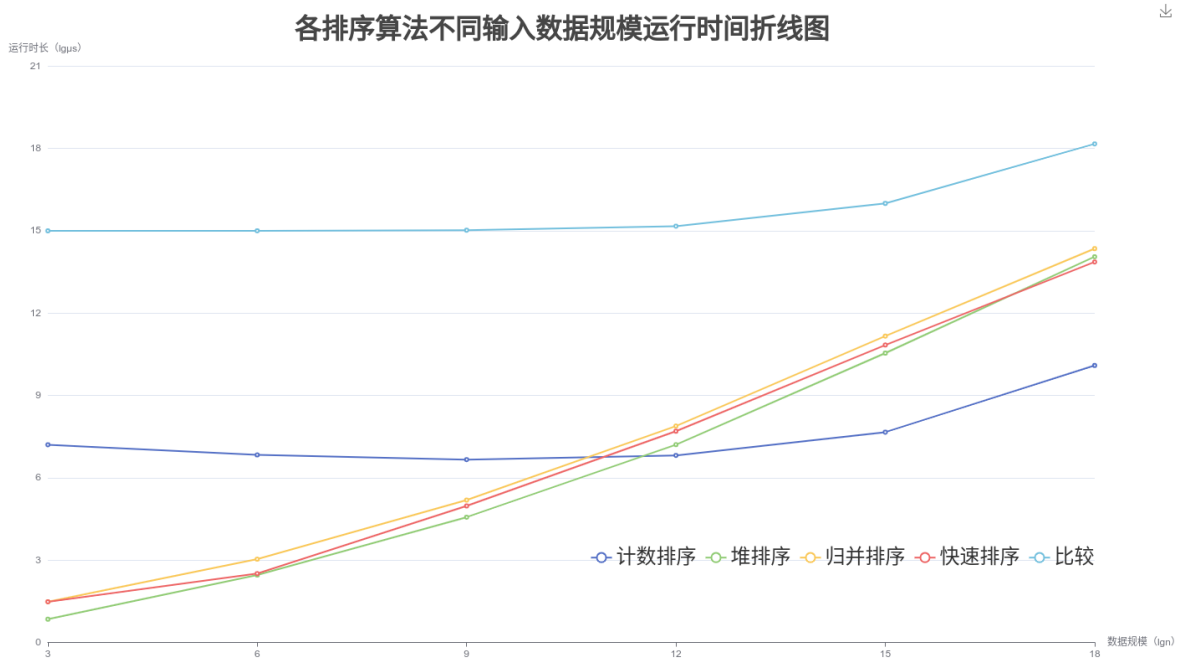



上图为 $y = x + \ln x$ 的图像，以做对比

由于上述计算中未考虑常数项（函数调用开销等），因此实际的运行时间初始增长会慢一点

对于计数排序，时间复杂度 $\Theta(n + k)$，计算得理论图像表达式：$y = \lg(2^x + k) = \lg(2^x + 2^{15})$

根据该表达式，增加一条辅助曲线：




趋势也是比较接近的，事实上，二者每点纵坐标之比为

```
[0.480423788095884, 0.45560908625246427, 0.443694096705882, 0.4493327227812122,
0.47903906592933104, 0.5555957578840631]
```

几乎为常数

- 初始时，$n$ 相对 $k$ 远小，因此用时几乎是一条直线
- 之后，因为 $k$ 固定，渐进复杂度符合 $\Theta(n)$

## 算法对比

从上面我们画的图中可以看到：

- 若只考虑基于比较的排序（可能排序对象不是 `int`）
  - 当 $n$ 比较小（$n < 2^{15}$）时，堆排序始终是最快的，快速排序稍慢，归并排序最慢
  - 当 $n$ 比较大（$n > 2^{18}$）时，快速排序明显成了最快的排序算法，其次是堆排序和归并排序
- 若还考虑计数排序
  - 当 $n$ 远小于值域时（例如本测试 $n < 2^{12}$），计数排序的性能不如上述基于比较的排序算法

- 当 $n$ 接近或远大于值域时，计数排序由于其 $\Theta(n+k)$ 的时间复杂度，是最快的