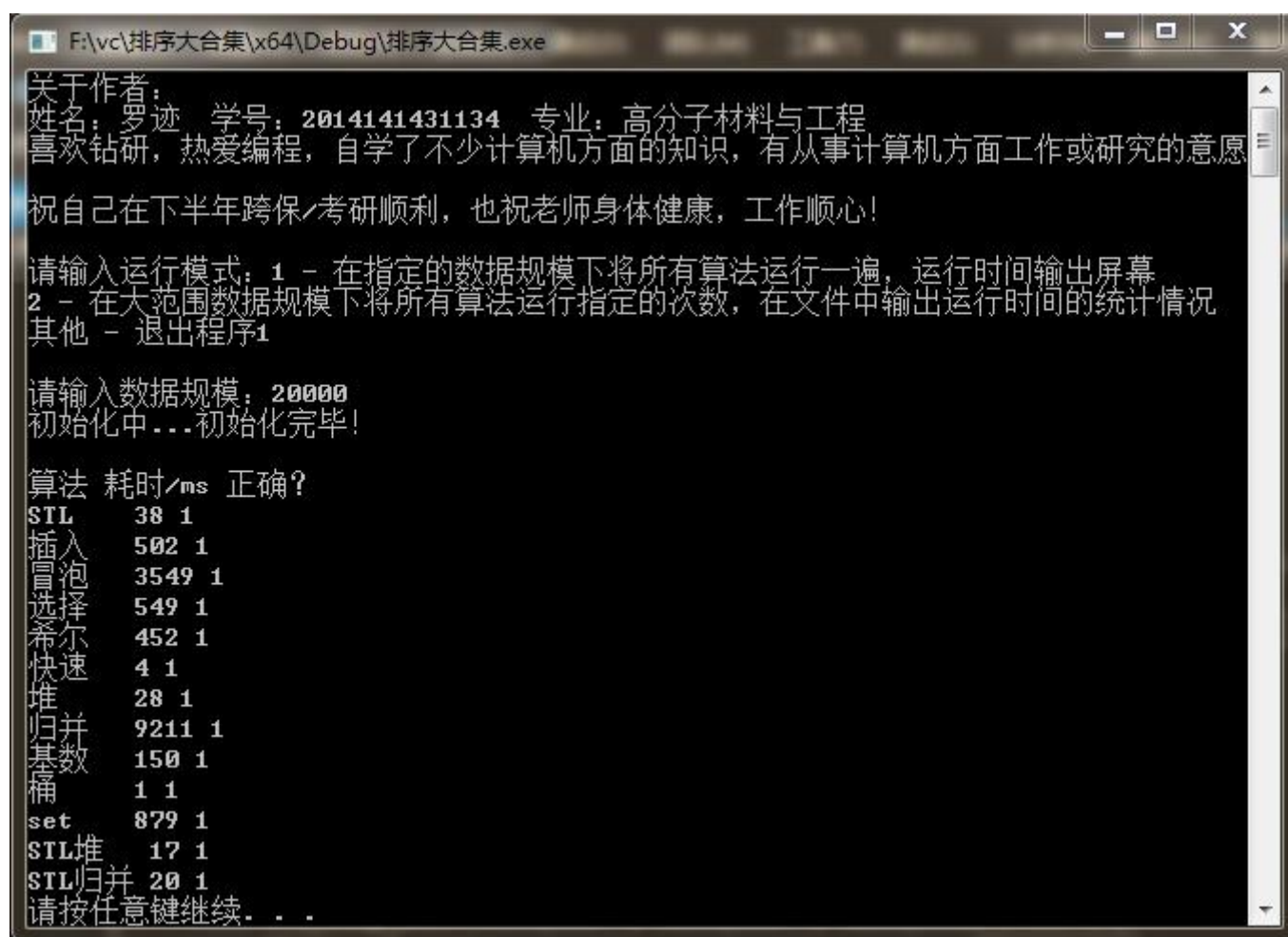


一、程序功能说明

本程序实现对十一大排序算法（插入排序 `insertion_sort`，冒泡排序 `bubble_sort`，选择排序 `selection_sort`，希尔排序 `shell_sort`，快速排序 `quick_sort`，堆排序 `heap_sort`，归并排序 `merge_sort`，基数排序 `radix_sort`，桶排序 `bucket_sort`，红黑树排序 `copy_to_set_sort`，内省排序 `intro_sort`）的模拟与统计。以上算法除红黑树排序利用 STL 中自带的 `set` 模板类（其内部结构为类似于平衡二叉排序树的红黑树），内省排序利用 STL 中自带的 `sort` 方法以外，**全部由作者本人亲自写出**。此外还对比了 STL 自带的堆排序（`make_heap` 和 `sort_heap` 方法）、归并排序（`stable_sort` 方法）与自己写的版本的性能差异。（本程序的排序指从小到大排序）

核心技术是排序算法和函数指针。

程序入口先显示自我介绍，通过 `about_myself` 类实现。之后有两个分支：按 1 输入要模拟的数据规模 `N`，程序随机产生 `N` 个 `0 ~ RAND_MAX`（通常为 32767）的整数储存在 `base_container` 类的对象 `all_nums_ori` 中，然后拷贝 12 份到 `container` 类的对象 `all_nums[12]` 中分别运行上述 13 种排序算法（内省排序直接在 `all_nums_ori` 上运行，运行结果用于检验另外 12 种排序算法的正确性），统计运行时间，检查结果正确性并输出。运行情况类似下图：



```
关于作者:
姓名: 罗迹 学号: 2014141431134 专业: 高分子材料与工程
喜欢钻研, 热爱编程, 自学了不少计算机方面的知识, 有从事计算机方面工作或研究的意愿

祝自己在下半年跨保/考研顺利, 也祝老师身体健康, 工作顺心!

请输入运行模式: 1 - 在指定的数据规模下将所有算法运行一遍, 运行时间输出屏幕
2 - 在大范围数据规模下将所有算法运行指定的次数, 在文件中输出运行时间的统计情况
其他 - 退出程序1

请输入数据规模: 20000
初始化中...初始化完毕!

算法 耗时/ms 正确?
STL      38 1
插入     502 1
冒泡    3549 1
选择     549 1
希尔     452 1
快速      4 1
堆        28 1
归并    9211 1
基数     150 1
桶         1 1
set      879 1
STL堆     17 1
STL归并   20 1
请按任意键继续. . .
```

图 1.1 模式 1 运行示例

按 2 将在如下图 `data_size` 所示规模的数据规模上运行测试。

```
unsigned* results[13][MAX_SZ_IDX]; //第一层是算法 (共13种), 第二层是运行规模 (共28个), 第三层是运行次数 (time次)
const unsigned data_size[] = {
    100, 200, 300, 400, 500, 600, 700, 800, 900,
    1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000,
    10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000
};
const string alg_names[12] = {
    "插入", "冒泡", "选择", "希尔", "快速",
    "堆", "归并", "基数", "桶", "set",
    "STL堆", "STL归并"
};
```

图 1.2 一些常量

输入对每个数据规模要运行的次数 `time`。每次运行都在同样的数据上将 13 种排序算法各运行一次；但下一次运行将重新产生随机数，以使测试涵盖尽可能广的数据分布，并保证对各算法的公平性。`time` 次运行完毕，将运行时间写入 `results` 三维数组中，将各算法在各数据规模上运行时间的平均结果输出到文件 `result.txt` 中，方便进行下一步

的图表分析。

二、程序代码解析

1. 首部

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <ctime>
#include <cmath>
#include <deque>
#include <set>
#include <algorithm>
#include <numeric>
#include <fstream>
#include <string>
#define MAX_SZ_IDX 28
using namespace std;
```

图 2.1.1 代码首部

编译环境 Visual Studio 2013 community。几点说明：

- 1) `_getche` 函数包含在 `conio.h` 中
- 2) 随机数产生器的初始化及运行计时需要用到时间，因此要包含 `ctime`
- 3) 希尔排序的增量序列采用目前公认的效率最高的 Sedgewick 序列（详细见下），产生该序列需要用到乘方函数，因此要包含 `cmath`
- 4) 在运行模式 2 下，最后统计运行时间的平均值用到了 `accumulate` 函数，这包含在 `numeric` 中
- 5) 宏 `MAX_SZ_IDX` 用于调试程序，其值最大为 28。在模式 2 下，程序对 `data_size[0] ~ data_size[MAX_SZ_IDX]` 的数据规模分别运行 `time` 次，`data_size` 数组最后的几个数据规模运行时间较长，不利于程序调试，因此在调试时改小 `MAX_SZ_IDX` 的值。

2. `about_myself` 类

```
class about_myself {
private:
    string name, ID, major, info, wish;
public:
    about_myself() {
        name = "姓名:罗迹 ";
        ID = "学号:2014141431134 ";
        major = "专业:高分子材料与工程";
        info = "喜欢钻研,热爱编程,自学了不少计算机方面的知识,有从事计算机方面工作或研究的意愿";
        wish = "祝自己在下半年跨保/考研顺利,也祝老师身体健康,工作顺心!";
    }
    void print() {
        cout << name << ID << major << endl;
        cout << info << endl;
        cout << wish << endl;
    }
};
```

图 2.2.1 `about_myself` 类

3. `base_container` 类

```

struct base_container {
    unsigned size;
    unsigned *begin, *end;
    base_container() {}
    base_container(const unsigned& N) { ... }
    base_container(const base_container& source) { ... }
    unsigned operator[](const unsigned& index) const { ... }
    void print() { ... }
    unsigned STL_sort() { ... }
};

```

图 2.3.1 base_container 类总览

由于所有成员均需设为公有，因此采用 struct 关键字。C++ 中 struct 和 class 的区别基本仅在于访问级别和继承方式的不同。

1) 构造函数

```

base_container(const unsigned& N) : size(N) {
    begin = new unsigned[N];
    end = begin + N;
    for (unsigned* i = begin; i < end; i++) *i = rand();
}

```

图 2.3.2 构造函数

参数为容器的大小。产生 N 个 0~32767 的随机整数，放入容器中。容器内部是一个数组，头指针 begin 指向第一个元素，尾指针 end 指向最后一个元素之后的第一个空位置，这样的设计与 STL 中的各大容器（vector、set、deque、map 等）一致。

2) 拷贝构造函数

```

base_container(const base_container& source) {
    size = source.size;
    begin = new unsigned[size];
    end = begin + size;
    for (unsigned i = 0; i < size; i++) *(begin + i) = source[i];
}

```

图 2.3.3 拷贝构造函数

由于类中包含指针，必须自行设计拷贝构造函数。这一函数用于其派生类 container 的构造（详细见下）。

3) []重载

```

unsigned operator[](const unsigned& index) const {
    if (index < size) return begin[index];
    else return RAND_MAX + 1;
}

```

图 2.3.4 []重载

返回索引值 index 指向的元素，当 index 超出容器容量时，返回 32768，这个值大于容器中所有可能出现的元素。这样的设计是专门为堆排序准备的（详细见下）。

4) print 方法

```

void print() {
    for (unsigned* i = begin; i < end; i++) cout << *i << ' ';
    cout << endl;
}

```

图 2.3.5 print 方法

为调试而设计的，在最终的程序中没有使用到。

5) 内省排序

```

    unsigned STL_sort() {
        time_t start, stop;
        start = clock();
        sort(begin, end);
        stop = clock();
        return stop - start;
    }

```

图 2.3.6 STL 提供的排序（内省排序）

这个排序算法直接在 `base_container` 类的对象上运行，接口与其他算法不同。`clock` 函数返回程序（准确说是进程）开始运行到此函数被调用所经过的时长（单位 `ms`）。因此，`STL_sort` 方法返回值为 `sort` 函数所运行的时长。**sort 函数是 C++ 提供的标准排序方法，内部实现为内省排序，实质是对快速排序的改良：当检测到快排递归深度过深时会停止快排转而使用堆排序，而当元素数目过少时则直接使用插入排序。**

4. container 类

在说明 `container` 类之前定义了全局变量 `all_nums_ori`，在 `container` 类中的 `check` 方法会使用到。

此类公有继承自 `base_container` 类，没有成员变量，仅对 `base_container` 类封装了一系列排序算法，因此仅设计了一个“拷贝”构造函数，直接调用基类的拷贝构造函数进行构造。

私有成员函数均用于堆排序。

```

base_container all_nums_ori;

class container : public base_container {
private:
    unsigned left_elem(unsigned idx) { ... }
    unsigned right_elem(unsigned idx) { ... }
    void heapify(unsigned idx) { ... }
    unsigned pop() { ... }
public:
    container() {}
    container(const base_container& source) : base_container(source) {}
    bool check() const { ... }
    unsigned runtime(void (container::*fp)(unsigned*, unsigned*, unsigned)) { ... }
    void copy_to_set_sort(unsigned*, unsigned*, unsigned) { ... }
    void insertion_sort(unsigned* b, unsigned*, unsigned interval) { ... }
    void bubble_sort(unsigned*, unsigned*, unsigned) { ... }
    void selection_sort(unsigned*, unsigned*, unsigned) { ... }
    void shell_sort(unsigned*, unsigned*, unsigned) { ... }
    void radix_sort(unsigned*, unsigned*, unsigned) { ... }
    void bucket_sort(unsigned*, unsigned*, unsigned) { ... }
    void quick_sort(unsigned* b, unsigned* e, unsigned) { ... }
    void merge_sort(unsigned* b, unsigned* e, unsigned) { ... }
    void heap_sort(unsigned*, unsigned*, unsigned) { ... }
    void STL_heap_sort(unsigned*, unsigned*, unsigned) { ... }
    void STL_merge_sort(unsigned*, unsigned*, unsigned) { ... }
};

```

图 2.4.1 container 类总览

1) check 方法

```

bool check() const {
    for (unsigned i = 0; i < size; i++)
        if (begin[i] != all_nums_ori.begin[i]) return false;
    return true;
}

```

图 2.4.2 check 方法

与 `STL_sort` 的结果进行逐一比对，检验排序结果的正确性。

2) runtime 方法


```

    unsigned runtime(void (container::*fp)(unsigned* , unsigned* , unsigned)) {
        time_t start, stop;
        start = clock();
        (this->*fp)(begin, end, 1);
        stop = clock();
        return stop - start;
    }

```

图 2.4.3 runtime 方法

参数为 container 类的 **函数指针**。用于对各排序算法进行计时，每次传入不同的排序算法对应的函数指针，返回传入函数的运行时间。要传入的函数指针统一存在如下的数组中：

```

void (container::*fp[12])(unsigned* , unsigned* , unsigned) = {
    &container::insertion_sort, &container::bubble_sort,
    &container::selection_sort, &container::shell_sort,
    &container::quick_sort, &container::heap_sort,
    &container::merge_sort, &container::radix_sort,
    &container::bucket_sort, &container::copy_to_set_sort,
    &container::STL_heap_sort, &container::STL_merge_sort
};

```

图 2.4.4 函数指针

下列排序算法的实现遵循两个原则：尽可能呈现其最原始最容易理解的面貌；尽可能使用指针或迭代器访问容器以提高效率。

3) 红黑树排序

```

void copy_to_set_sort(unsigned* , unsigned* , unsigned) {
    multiset<unsigned> set_container;
    for (unsigned* i = begin; i < end; i++) set_container.insert(*i);
    unsigned* j = begin;
    for (auto num : set_container) *(j++) = num;
}

```

图 2.4.5 红黑树排序

set 内部维护了一棵红黑树，multiset 在其基础上允许元素重复。将容器中的整数逐一拷贝入 multiset 中，multiset 内部会自动排好序（如同在二叉排序树中逐一插入元素一样），然后再用 multiset 内部的有序序列覆盖原容器即可。本方法原本不需要任何参数，留三个空参数只是为了接口的一致性，使得 runtime 方法能够接受其函数指针。以下的算法类似。

4) 插入排序

```

void insertion_sort(unsigned* b, unsigned* , unsigned interval) {
    for (unsigned* i = b; i < end - interval; i += interval)
        if (*i > *(i + interval)) { //找到无序序列的第一个元素*(i + interval)
            unsigned* j = b;
            for (; *j <= *(i + interval); j += interval); //找到该元素应插入的位置j
            unsigned temp = *(i + interval); //保存待插入元素
            for (unsigned* t = i; t >= j; t -= interval) //j及之后的有序序列元素后移
                *(t + interval) = *t;
            *j = temp; //在j位置插入temp
        }
}

```

图 2.4.6 插入排序

两个有名参数原本也是不需要的，在这里供希尔排序调用时使用。b 指待排序序列的起始位置（对于普通的插入排序就是 begin），interval 指两元素之间的跨度（对于普通的插入排序就是 1）。

5) 冒泡排序

```

void bubble_sort(unsigned* , unsigned* , unsigned) {
    for (unsigned* i = end; i > begin; i--)
        for (unsigned* j = begin; j < i; j++)
            if (*j > *(j + 1)) swap(*j, *(j + 1));
}

```

图 2.4.7 冒泡排序

6) 选择排序

```

void selection_sort(unsigned* , unsigned* , unsigned) {
    for (unsigned* j = end; j > begin; j--) {
        unsigned* max = begin;
        for (unsigned* i = begin; i <= j; i++) //寻找begin ~ j位置的最大值
            if (*i > *max) max = i;
        swap(*max, *j); //将找到的最大值换到最后
    }
}

```

图 2.4.8 选择排序

7) 希尔排序

```

void shell_sort(unsigned* , unsigned* , unsigned) {
    //Sedgewick增量序列公式:  $9 * 4^i - 9 * 2^i + 1$  (i从0开始) 或  $4^i - 3 * 2^i + 1$  (i从2开始) (^表示乘方)
    set<unsigned> Sedgewick_seq; //增量序列需要按序排列, 因此使用set存放
    unsigned elem = 1;
    //当某一个增量值不小于序列总长度的一半, 结束计算
    for (unsigned i = 0; elem < size / 2; i++) {
        unsigned pow_2 = pow(2, i);
        elem = 9 * pow_2 * (pow_2 - 1) + 1;
        if (elem < size / 2) Sedgewick_seq.insert(elem);
    }
    for (unsigned i = 2; elem < size / 2; i++) {
        unsigned pow_2 = pow(2, i);
        elem = pow_2 * (pow_2 - 3) + 1;
        if (elem < size / 2) Sedgewick_seq.insert(elem);
    }
    //以上为Sedgewick增量序列的计算
    //set默认从小到大排列, 因此需要逆序遍历Sedgewick序列, 使用rbegin/rend逆序迭代器
    for (auto i = Sedgewick_seq.rbegin(); i != Sedgewick_seq.rend(); i++)
        for (unsigned j = 0; j < *i; j++)
            insertion_sort(begin + j, end, *i);
}

```

图 2.4.9 希尔排序

许多算法将增量序列写成一个常量数列来使用, 对此笔者并不认同。笔者认为: 其一, 增量序列的选取是希尔排序重要的一步, 在计时中不应被忽略; 其二, 对于足够大的数据规模, 我们是无法预知所需的最大增量值的, 必须通过计算。因此将 Sedgewick 序列的获取写在了 shell_sort 方法内部。

8) 基数排序

```

void radix_sort(unsigned* , unsigned* , unsigned) {
    deque<unsigned> buckets[10]; //根据基数分解待排正整数, 按某一位上的值 (0..9) 将元素撒入十个桶中
    //待排正整数的取值范围为0 ~ 32767, 最多有5位, i == 0 代表当前处理的是个位, 将5位全部处理完毕则序列排序完毕
    for (unsigned i = 0; i < 5; i++) {
        for (unsigned* p = begin; p < end; p++) {
            unsigned radix = unsigned(*p / pow(10, i)) % 10; //取得基数
            buckets[radix].push_back(*p); //撒入相应的桶中
        }
        unsigned* j = begin;
        for (unsigned k = 0; k < 10; k++) //收集桶中数据, 覆盖原序列
            for (; !buckets[k].empty(); j++) {
                *j = buckets[k].front();
                buckets[k].pop_front();
            }
    }
}

```

图 2.4.10 基数排序

9) 桶排序


```

void bucket_sort(unsigned* , unsigned* , unsigned) {
    unsigned* buckets = new unsigned[RAND_MAX + 1](); //申请32768个桶, 每个桶中存放对应元素出现的次数
    for (unsigned* i = begin; i < end; i++) buckets[*i]++; //统计各元素出现次数
    unsigned* temp_ptr = begin;
    for (unsigned i = 0; i < RAND_MAX + 1; i++) //利用桶中的信息覆盖写入原容器
        for (unsigned j = 1; j <= buckets[i]; j++)
            *(temp_ptr++) = i;
    delete[] buckets; buckets = NULL;
}

```

图 2.4.11 桶排序

第一句使用 `new` 而不直接写作 `unsigned buckets[RAND_MAX + 1] = {0};` 的原因是后者会导致栈内存使用过多。C++ 将内存划分为三个逻辑区域：堆、栈和静态存储区。使用一般方法（即不使用 `new` 或者 `malloc/calloc`）定义的局部变量会使用栈内存，而栈内存一般是比较小的（1~2M），编译器建议我**大小超过 1024 的数组不要使用栈内存**。使用 `new` 或 `malloc/calloc` 方法定义变量则会申请堆内存，堆内存足够大，但需要程序员手动管理（使用 `delete` 或 `free`），且读取较栈内存耗时。静态存储区则用来存放静态变量和全局变量。

10) 快速排序

```

void quick_sort(unsigned* b, unsigned* e, unsigned) {
    //e指向待排序序列最后一个元素的后一位
    if (e - b == 2) { //这意味着待排序序列长度为2
        if (*b > *(e - 1)) swap(*b, *(e - 1));
    }
    else if (e - b == 1 || e - b == 0) return;
    //以上定义递归终点
    else {
        unsigned pivot = *(e - 1); //选取最后一个元素为轴值
        unsigned* low = b; unsigned* high = e - 2;
        while (low < high) {
            for (; *low <= pivot && low < e - 1; low++); //从low向右找到第一个比轴值大的元素
            for (; *high >= pivot && high > b; high--); //从high向左找到第一个比轴值小的元素
            if (low < high) swap(*low, *high); //若low、high没有错位, 交换它们所指向的值
        }
        swap(*low, *(e - 1)); //将*low与序列最后一位的轴值交换
        quick_sort(b, low, 1); //递归排左边
        quick_sort(low + 1, e, 1); //递归排右边
    }
}

```

图 2.4.12 快速排序

11) 归并排序

```

void merge_sort(unsigned* b, unsigned* e, unsigned) {
    if (e - b == 1) return; //e指向待排序序列最后一个元素的后一位
    else if (e - b == 2) { //这意味着待排序序列长度为2
        if (*b > *(e - 1)) swap(*b, *(e - 1));
    } //以上定义递归终点
    else {
        merge_sort(b, b + (e - b) / 2, 1); //递归排左边
        deque<unsigned>* left_array = new deque<unsigned>(b, b + (e - b) / 2); //将排序结果存入一个deque中
        merge_sort(b + (e - b) / 2, e, 1); //递归排右边
        deque<unsigned>* right_array = new deque<unsigned>(b + (e - b) / 2, e); //将排序结果存入一个deque中
        deque<unsigned>* merged_array = new deque<unsigned>; //左右归并的结果放在这个deque中
        //比较左右两个deque的头元素, 较小的那个从它所在的deque中抛出并放入merged_array, 直到某一个deque为空
        while (!left_array->empty() && !right_array->empty())
            if (left_array->front() < right_array->front()) {
                merged_array->push_back(left_array->front());
                left_array->pop_front();
            }
            else {
                merged_array->push_back(right_array->front());
                right_array->pop_front();
            }
        //将还未抛空的deque接到merged_array最后
        if (!left_array->empty()) merged_array->insert(merged_array->end(), left_array->begin(), left_array->end());
        if (!right_array->empty()) merged_array->insert(merged_array->end(), right_array->begin(), right_array->end());
        //用merged_array覆盖原序列
        unsigned* j = b;
        for (auto num : *merged_array) *(j++) = num;
        delete left_array; delete right_array; delete merged_array;
    }
}

```

图 2.4.13 归并排序

为方便截图，代码格式调整得较为紧凑。

使用 deque（双端队列，内部是小块数组用指针相连的链表，是数组与链表的折中）的原因在于归并排序左右两个序列需要频繁地在一端删除，在另一端增加元素。本来最好使用 list（内部是一个单个元素串接的链表），但实验发现 list 的效率非常低（这可能是由于 list 的析构是逐个元素逐个元素地析构的，在数据量非常庞大的情况下效率很低），遂改用 deque，发现效率有所提升，但依然不可接受（详见下）。

12) 堆排序

```
private:
    //以下两个函数，若对应节点不存在，则返回RAND_MAX + 1(32768)
    unsigned left_elem(unsigned idx) {
        return (*this)[idx * 2 + 1];
    }
    unsigned right_elem(unsigned idx) {
        return (*this)[idx * 2 + 2];
    }
    void heapify(unsigned idx) { //调整堆化
        //建立最小值堆，若某节点的左右两子节点的值均不小于自己，则无需调整
        if ((*this)[idx] <= left_elem(idx) && (*this)[idx] <= right_elem(idx));
        else { //不满足堆的性质，需要交换
            if (right_elem(idx) <= left_elem(idx)) {
                //若右孩子不大于左孩子，则将右孩子与其父节点交换
                swap(begin[idx * 2 + 2], begin[idx]);
                heapify(idx * 2 + 2); //递归地对右孩子的位置调整堆化
            }
            else { //若右孩子大于左孩子，则将左孩子与其父节点交换
                swap(begin[idx * 2 + 1], begin[idx]);
                heapify(idx * 2 + 1); //递归地对左孩子的位置调整堆化
            }
        }
    }
    unsigned pop() { //弹出堆顶元素
        unsigned res = *begin; //保存堆顶元素
        *begin = RAND_MAX + 1; //将堆顶元素置为32768（相当于让它不存在，那么在堆化时它会被压到堆底）
        heapify(0); //从堆顶调整堆化
        return res; //返回保存的值
    }
}
```

图 2.4.14 堆排序辅助函数

笔者采用了一种更容易理解的方式实现堆排序，虽然效率稍低于通常的方式。将待排序列拷贝出来建立最小值堆，而后每次将堆顶元素（即当前待排序列的最小元素）抛出，填入原待排序列相应的位置。抛出堆顶元素后从堆顶的位置调整堆化，保证下一次又抛出剩余元素中最小的那一个。相当于优化了简单选择排序中选择最小值的那一步。

对 heapify 方法的进一步说明：

在左右两子节点至少有一个的值小于父节点（即能够进入第一层 else）的情况下进行分类讨论：

A. 左 \geq 右，则左右两孩子必然都存在（注意到堆是一棵完全二叉树，不可能出现某节点左孩子不存在而右孩子存在的情况；且 heapify 方法不会传入一个叶子的索引值，因此也不存在左右两孩子都不存在的情况），那么右孩子必然小于父节点，应将其与父节点交换

B. 左 $<$ 右。那么，不可能将较大或实际不存在（表现为 32768）的右孩子与其父节点交换。又因为左右两孩子至少有一个比父节点小，所以左孩子一定小于父节点，应将其与父节点交换

```
void heap_sort(unsigned* , unsigned* , unsigned) {
    container temp_array = *this; //将待排序列拷贝出来
    unsigned last_parent_idx = (size - 2) / 2; //找到最后一个非叶子节点的索引值
    for (int i = last_parent_idx; i >= 0; i--) temp_array.heapify(i); //初次调整建堆
    //逐个抛出堆顶元素并填入原待排序列
    for (unsigned* i = begin; i < end; i++) *i = temp_array.pop();
}
```

图 2.4.15 堆排序具体实现

13) STL 提供的堆排序和归并排序


```

void STL_heap_sort(unsigned* , unsigned* , unsigned) {
    make_heap(begin, end);
    sort_heap(begin, end);
}

void STL_merge_sort(unsigned* , unsigned* , unsigned) {
    stable_sort(begin, end);
}

```

图 2.4.16 STL 提供的堆排序和归并排序

5. 模式 1 所用的 run_once_print 函数

```

void run_once_print() {
    cout << endl << "请输入数据规模:";
    unsigned N; cin >> N;
    cout << "初始化中...";
    all_nums_ori = base_container(N); //构造待排序序列
    container all_nums[12];
    for (unsigned i = 0; i < 12; i++) all_nums[i] = base_container(all_nums_ori); //拷贝12份
    cout << "初始化完毕!" << endl << endl;
    cout << "算法 耗时/ms 正确?" << endl;
    //输出STL_sort (内省排序) 的运行时长
    cout << "STL      " << all_nums_ori.STL_sort() << ' ' << '1' << endl;
    for (unsigned i = 0; i < 12; i++) {
        //对拷贝出的12份数据运行另外12个排序函数, 计时, 输出
        cout << alg_names[i] << all_nums[i].runtime(fp[i]) << ' ' << endl;
        cout << all_nums[i].check() << endl; //检查排序结果是否正确
    }
}

```

图 2.4.17 run_once_print 函数

6. 模式 2 所用的 run_once 函数

```

void run_once(unsigned sz_idx, unsigned t) { //参数1指明要运行的数据规模, 参数2指明当前运行的是第几次
    all_nums_ori = base_container(data_size[sz_idx]); //构造待排序序列
    container all_nums[12];
    for (unsigned i = 0; i < 12; i++) all_nums[i] = base_container(all_nums_ori); //拷贝12份
    results[0][sz_idx][t] = all_nums_ori.STL_sort(); //对原始数据运行STL_sort (内省排序) 并计时
    for (unsigned i = 0; i < 12; i++) //对拷贝出的12份数据运行另外12个排序函数并计时
        results[i + 1][sz_idx][t] = all_nums[i].runtime(fp[i]);
}

```

图 2.4.18 run_once 函数

7. main 函数

```

int main() {
    cout << "关于作者:" << endl;
    about_myself().print();
    srand(time(0)); //初始化随机数生成器
    cout << endl << "请输入运行模式: 1 - 在指定的数据规模下将所有算法运行一遍, 运行时间输出屏幕" << endl;
    cout << "2 - 在大范围数据规模下将所有算法运行指定的次数, 在文件中输出运行时间的统计情况 其他 - 退出程序";
    char m = _getche(); cout << endl;
    switch (m) {
        case '1': run_once_print(); break;
    }
}

```

图 2.4.19 main 函数第一部分

```

case '2': {
    cout << "请输入运行次数, 程序将对 100 - 100000 的数据规模进行运行, 在result.txt中输出各算法在不同数据规模上运行时间";
    unsigned time; cin >> time;
    for (unsigned i = 0; i < 13; i++) //results三维数组初始化
        for (unsigned j = 0; j < MAX_SZ_IDX; j++)
            results[i][j] = new unsigned[time];
    //对data_size中各大数据规模运行time次, 耗时记录在results中
    for (unsigned i = 0; i < MAX_SZ_IDX; i++)
        for (unsigned j = 0; j < time; j++) {
            run_once(i, j);
            cout << "数据规模" << data_size[i] << "运行第" << j + 1 << "次结束!" << endl;
        }
    //在文件中输出统计结果
    ofstream file("result.txt");
    for (unsigned i = 0; i < 13; i++)
        for (unsigned j = 0; j < MAX_SZ_IDX; j++) {
            if (i == 0) file << "STL";
            else file << alg_names[i - 1];
            file << ' ' << data_size[j] << ' '
                << accumulate(results[i][j], results[i][j] + time, 0) / time << endl;
            //accumulate函数对results[i][j][0]到results[i][j][time - 1]求和
            delete[] results[i][j];
        }
    file.close(); break;
}
default: break;
}
system("pause");
return 0;
}

```

图 2.4.20 main 函数第二部分

三、实验结果分析

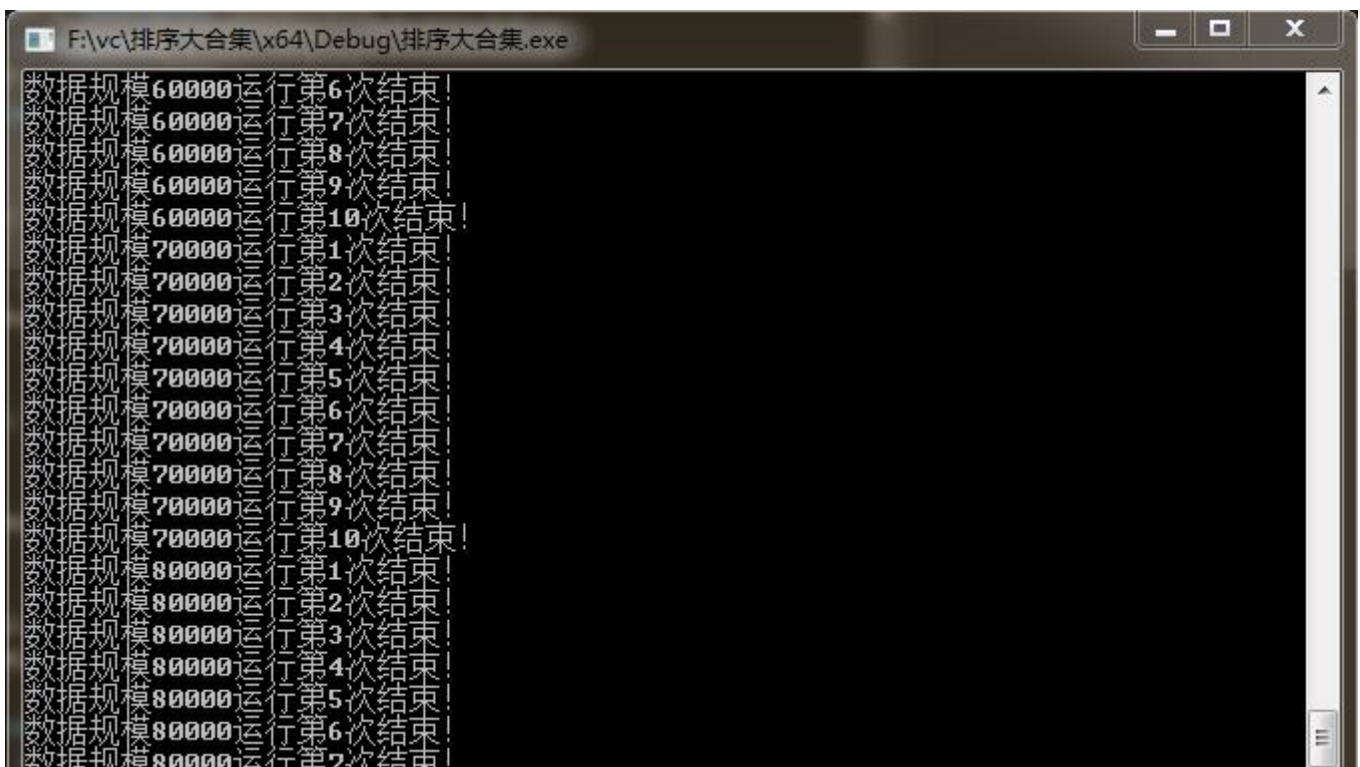


图 2.4.17 模式 2 运行示例

在模式 2 下, 对图 1.2 中所有数据规模运行 10 次, 统计结果如下:

	A	B	C
1	STL	100	0
2	STL	200	0
3	STL	300	0
4	STL	400	0
5	STL	500	0
6	STL	600	0
7	STL	700	0
8	STL	800	0
9	STL	900	0
10	STL	1000	0
11	STL	2000	1
12	STL	3000	2
13	STL	4000	3
14	STL	5000	4
15	STL	6000	5
16	STL	7000	6
17	STL	8000	7
18	STL	9000	8
19	STL	10000	11
20	STL	20000	20
21	STL	30000	32
22	STL	40000	43
23	STL	50000	54
24	STL	60000	66
25	STL	70000	78
26	STL	80000	89
27	STL	90000	102
28	STL	100000	112
29	插入	100	0
30	插入	200	0
31	插入	300	0
32	插入	400	0

图 3.1 输出的 txt 导入 excel 表格

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	数据规模	内省	插入	冒泡	选择	希尔	快速	堆	归并	基数	桶	set	STL堆	STL归并
2	100	0	0	0	0	0	0	0	48	3	0	5	0	0
3	200	0	0	0	0	0	0	0	104	5	0	9	0	0
4	300	0	0	0	0	0	0	0	137	6	0	12	0	0
5	400	0	0	1	0	0	0	0	163	6	0	14	0	0
6	500	0	0	2	0	0	0	0	166	7	0	17	0	0
7	600	0	0	2	0	0	0	0	186	7	0	20	0	0
8	700	0	0	3	0	0	0	0	254	9	0	23	0	0
9	800	0	0	4	0	1	0	1	248	9	0	25	0	0
10	900	0	1	6	1	1	0	0	217	8	0	24	0	0
11	1000	0	1	7	1	1	0	1	208	9	0	30	0	0
12	2000	1	5	31	5	4	0	2	220	18	0	37	1	1
13	3000	2	11	72	12	9	0	3	570	38	0	40	2	2
14	4000	3	19	127	21	17	0	4	872	64	0	67	2	3
15	5000	4	30	203	33	27	1	6	1433	94	0	99	3	4
16	6000	5	43	291	48	38	1	7	2307	134	0	141	4	5
17	7000	6	59	399	65	52	1	8	2924	173	0	182	5	6
18	8000	7	77	522	85	67	1	10	2551	173	0	185	6	7
19	9000	8	98	664	108	84	1	11	2085	171	0	185	7	8
20	10000	11	122	826	133	103	2	13	1567	161	0	178	8	8
21	20000	20	501	3363	533	413	4	27	8876	554	0	596	16	19
22	30000	32	1127	7620	1197	917	6	43	5785	423	0	543	26	29
23	40000	43	2006	13628	2125	1623	9	59	6752	278	0	448	36	40
24	50000	54	3140	21197	3319	2526	11	76	8486	343	0	607	45	51
25	60000	66	4523	30494	4778	3633	14	94	9212	407	0	827	55	61
26	70000	78	6164	41580	6504	4940	16	109	10737	486	0	1227	65	75
27	80000	89	8041	54172	8485	6470	19	127	13712	565	0	1456	76	85
28	90000	102	10172	68624	10734	8171	21	145	16311	636	1	1775	86	96
29	100000	112	12570	84908	13264	10088	24	162	18818	716	1	2262	96	108

图 3.2 制表（txt 中的 STL 指内省排序）

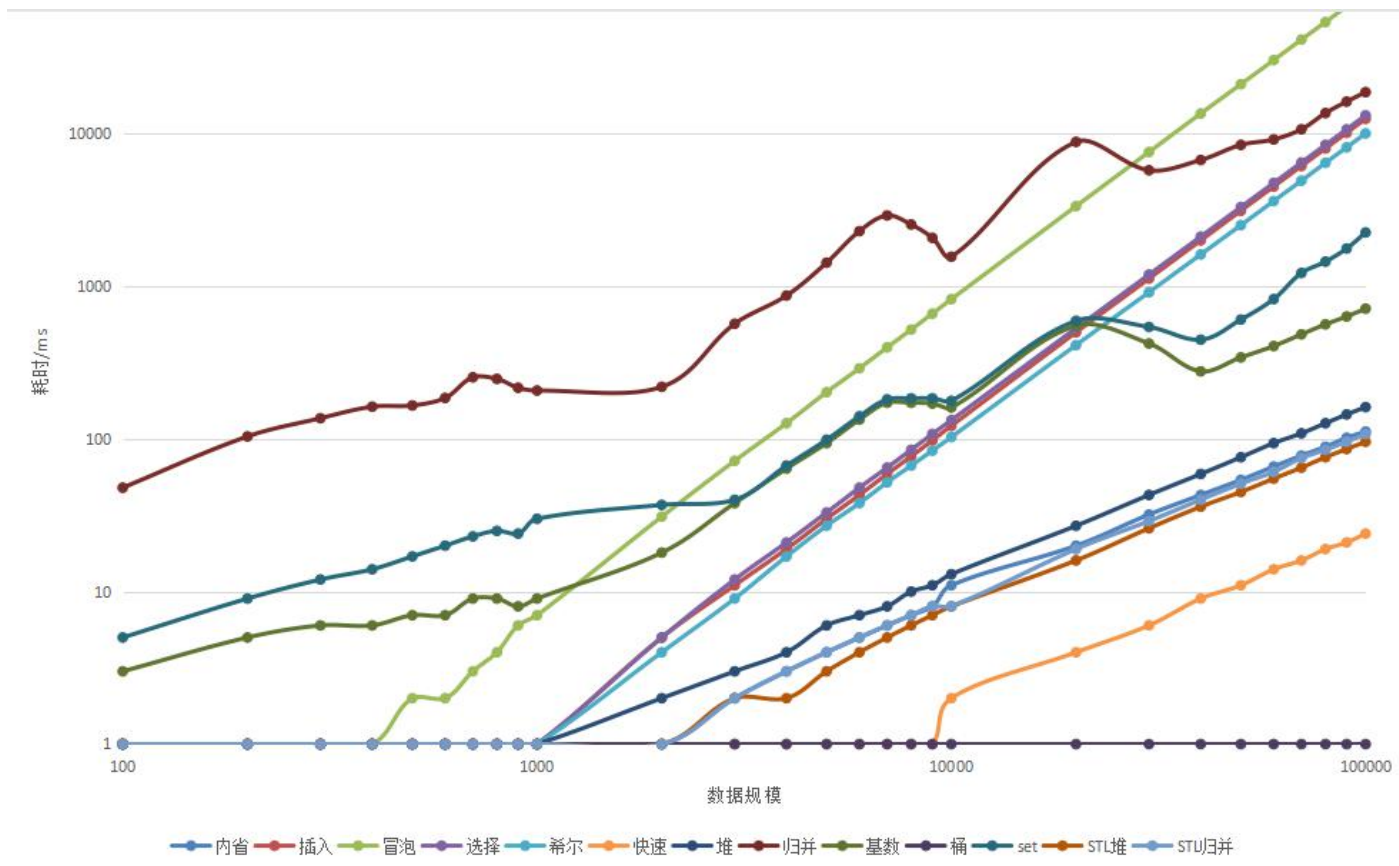


图 3.3 在双对数坐标下制作汇总图

由总图我们可以将排序算法分为三类：

1. “三大奇”：基数排序、红黑树排序、笔者写的归并排序

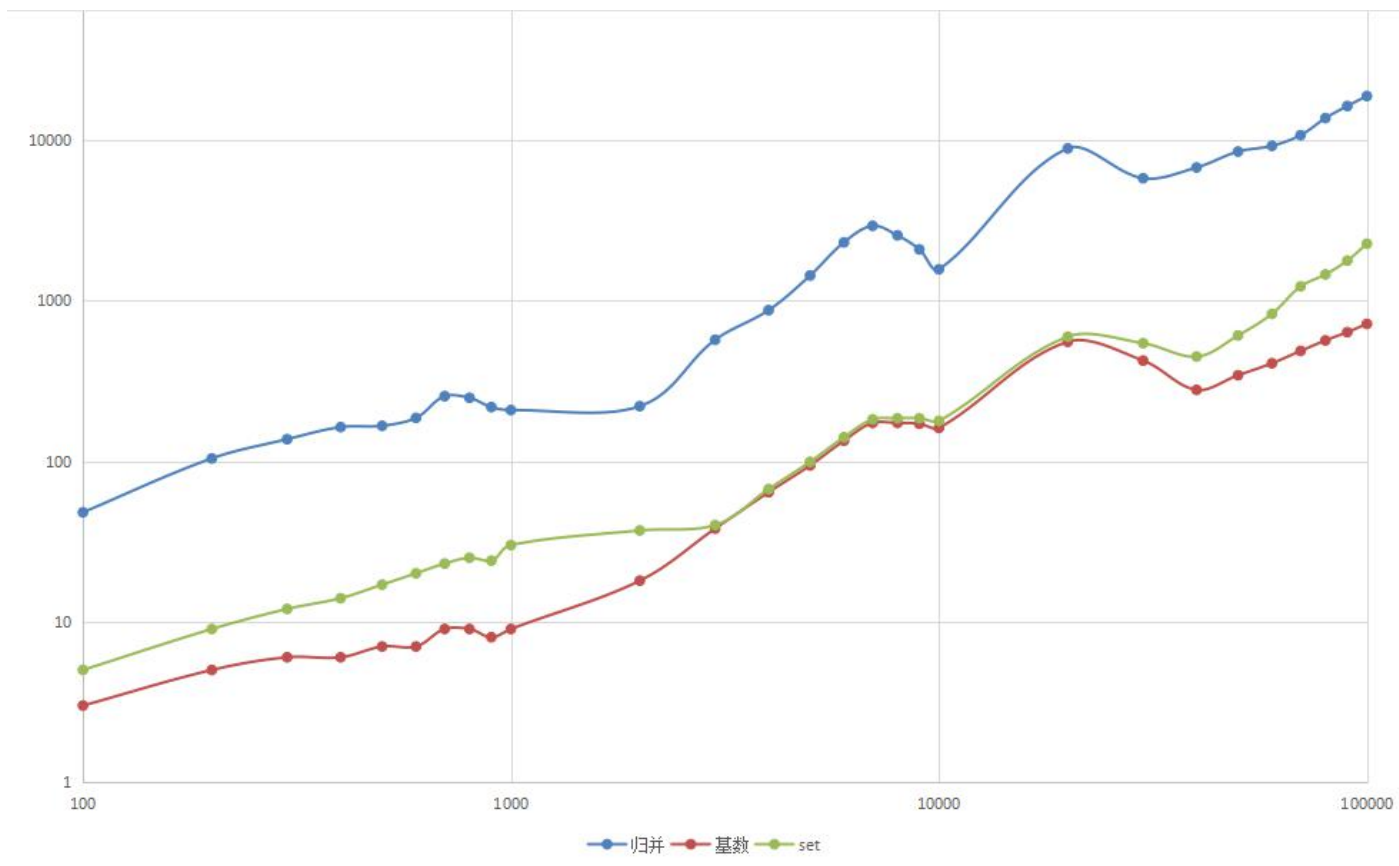


图 3.4 “三大奇”（双对数坐标）

这三种算法的表现十分奇怪，在数据规模很小的时候也需要花相当的时间（甚至比冒泡排序还慢），但增长速率缓慢，呈“阶梯式”，中途有陡增陡降（从 7000 ~ 10000, 20000 ~ 40000 都表现出耗时减小，而在下降之前都有突增）。

而且它们的变化趋势相近。研究它们的代码可以发现，它们都使用了 STL 容器（基数排序和归并排序使用了 `deque`，红黑树排序使用了 `multiset`）。开辟和析构这些容器大概花费了太多的时间，这使得它们在数据量很小的情况下表现十分糟糕。而中途的下降可能与 STL 容器的内存分配方式有关，数据规模每跨过一个“门槛”（如 10000），在向 STL 容器中添加数据时会涉及内存的重新分配，这消耗了大量的时间，而跨过门槛后又逐渐趋于稳定，因为要到下一个“门槛”才会有新一轮的重新分配。

2. “四大慢”：冒泡排序、插入排序、选择排序、希尔排序

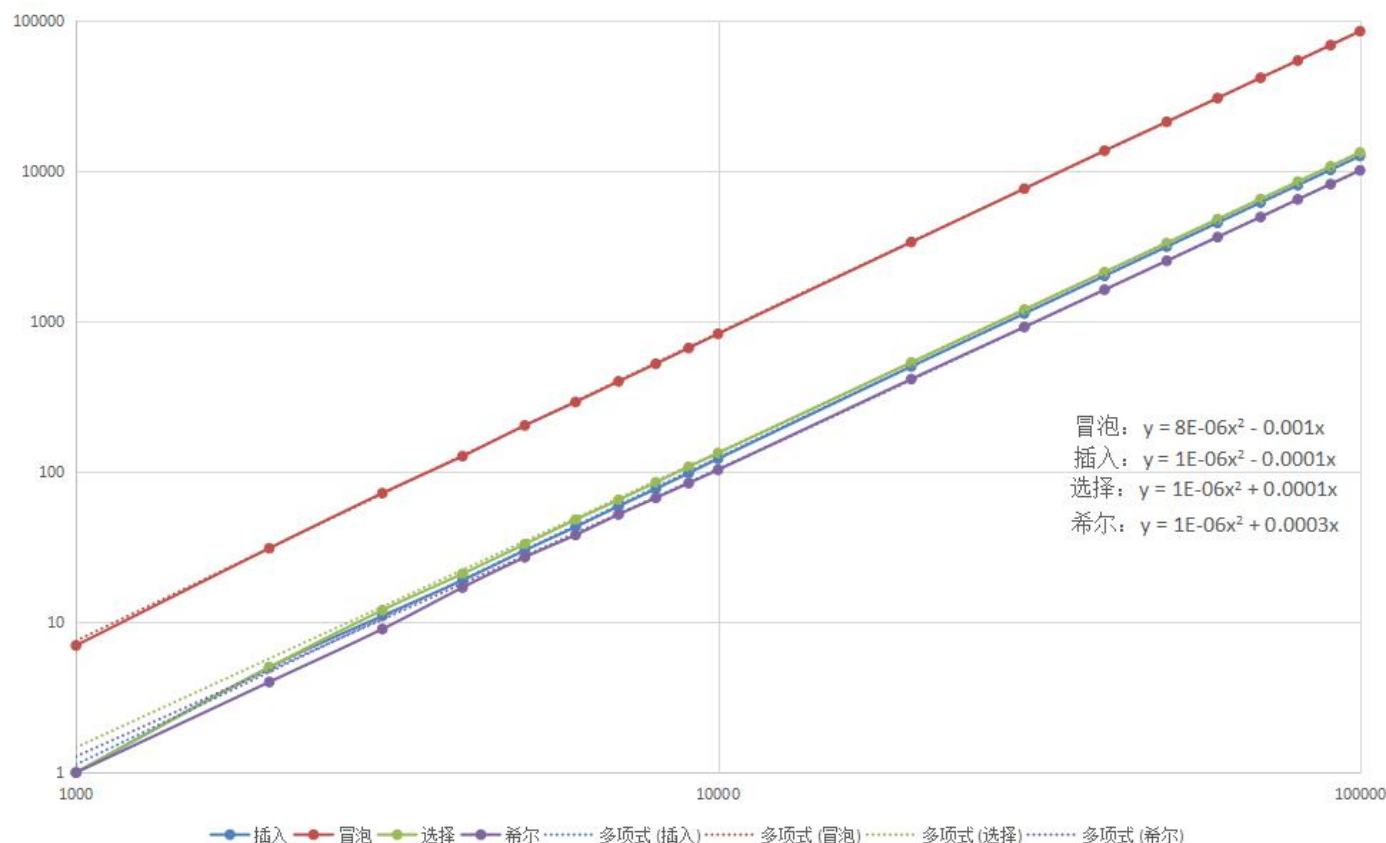


图 3.5 “四大慢”（双对数坐标）

冒泡、选择、插入排序的表现与理论基本符合，耗时对数据规模呈现二次关系。但希尔排序似乎没有数量级上的优化，直接从表格观察，在数据量足够大时其耗时大致是直接插入排序的 80 %。这匪夷所思，笔者未能找到一个合适的解释。

3. “五大快”：堆排序、快速排序、内省排序、STL 提供的堆排序和归并排序

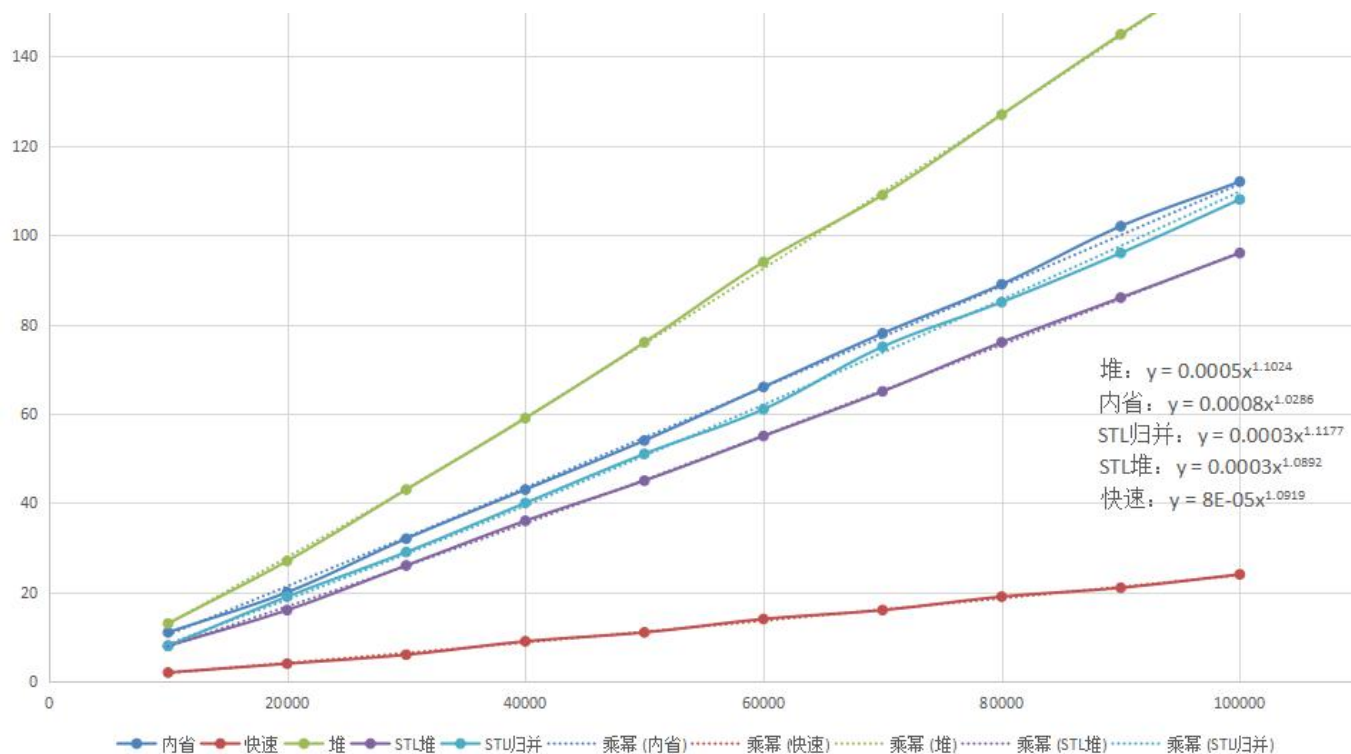


图 3.6 “五大快”（未使用双对数坐标，数据规模从 10000 开始）

这几种排序在理论上是最为优秀的，时间代价 $O(n \log n)$ 。Excel 无法根据 $n \log n$ 这样的形式进行拟合，因此笔者选用了幂函数。快速排序可谓实至名归，占据了压倒性的优势；反而 STL 提供的，更优化的内省排序没能表现出优势。但是从拟合结果我们可以看到，内省排序 x 的幂次是 1.0286，几乎接近于线性，优于快排的 1.0919。这也就是说，在数据量更大时，快速排序会被内省排序超越。

笔者自己写的堆排序不如 STL 提供的堆排序快，这也在意料之中；归并排序就更是如此。有空可以学习一下 STL 源码，提高自己的代码设计能力。