Project1：排序

PB15111672 林郅琦

实验要求：

- **实验1**：排序 n 个元素，元素为随机生成的长为1..32的字符串（字符串均为英文小写字母），n 的取值为：22，25，28，211，214，217；算法：直接插入排序，堆排序，归并排序，快速排序。
- **实验2**：排序n个元素，元素为随机生成的1到65535之间的整数，n的取值为： 22，25，28，211，214，217 ； 算法：冒泡排序，快速排序， 基数排序，计数排序。
- 字符串排序大小判断标准：
    - 首先按字符串长度进行排序（短字符串在前，长字符串在后）。
    - 对长度相同的字符串，按字母顺序进行排序。

实验环境

- 操作系统: MacOS High Sierra
- 编译环境: g++ ; Apple LLVM version 9.0.0 (clang-900.0.37)
- 机器内存: 8G
- 时钟主频: 处理器主频: 2.9GHz; 内存主频: 2133 MHz

实验过程

1. 生成随机数据
    - 生成随机整数: 只要使用 C++ 中 rand() 函数进行生成即可，每生成一个随机数，就将其写入数据产生文件，一共生成 2^17 个随机整数，每个整数占据一行位置。
    - 生成随机字符串
        - 先生成1~32范围内的随机整数，此整数为此次随机生成字符串的长度，接下来通过 std::string 遍历随机生成小写字母组成字符串，作为一个随机生成的字符串。
        - 没生成一个字符串，便将其写入文件，每个字符串占据1行，共生成 2^17 个字符串。
2. 排序算法
    - 几乎所有排序算法采用 C++ 模板类进行实现（个别只有特定数据类型适用的排序算法，如基数排序、计数排序除外），并引入比较函数函数指针作为对应数据类型的比较标准，排序算法接口基本一致，采用以下接口:

```
template<typename T>
void SortAlgorithm(T* array, int len, bool (*Compare)(T&, T&));
```

其中：T 为数据类型，array 为对应排序数组，len 为排序数组长度，Compare 为比较函数函数指针。
排序算法可参照《算法导论》进行实现。

3. 其他过程如数据读取、数据存储过于简单，不再描述。

实验关键代码

1. 生成随机字符串:

```
string RandomString(int len){
    int strsize = rand()%(32) + 1; // 产生随机字符串长度
    string str;
    str.resize(strsize);
    for(int i = 0; i < strsize; ++i)
        str[i] = (char)lzq::RandomInt('a','z'); //产生'a' - 'z' 的随机字符
    return str;
}
```

2. 排序算法：
   所有排序算法均在 sort.h 中，实现过程可自行查看
3. 数据读取:

```
// 每个算法调用前均会调用这个函数
// datatype 对排序数据的选择：字符串或者整数
// alg 为排序的算法名称
// len 表示排序数据的长度
void GetData(string datatype, string alg, size_t len){
    if(datatype == string("str")){
        datafile = ifstream("../input/input_strings.txt");
        for(int i = 0; i < len; ++i)
            getline(datafile, str[i]);
    }
    else if(datatype == string("int")){
        datafile.open("../input/input_ints.txt");
        for(int i = 0; i < len; ++i)
            datafile >> intarray[i];
    }
    else{
        cerr << "Unknown DataType" << endl;
        exit(1);
    }
    datafile.close();
    cout << "Start Sorting: " << "Algorithm: " << alg << endl;
    timer.StartCount(); //开始算法计时
}
```

实验结果

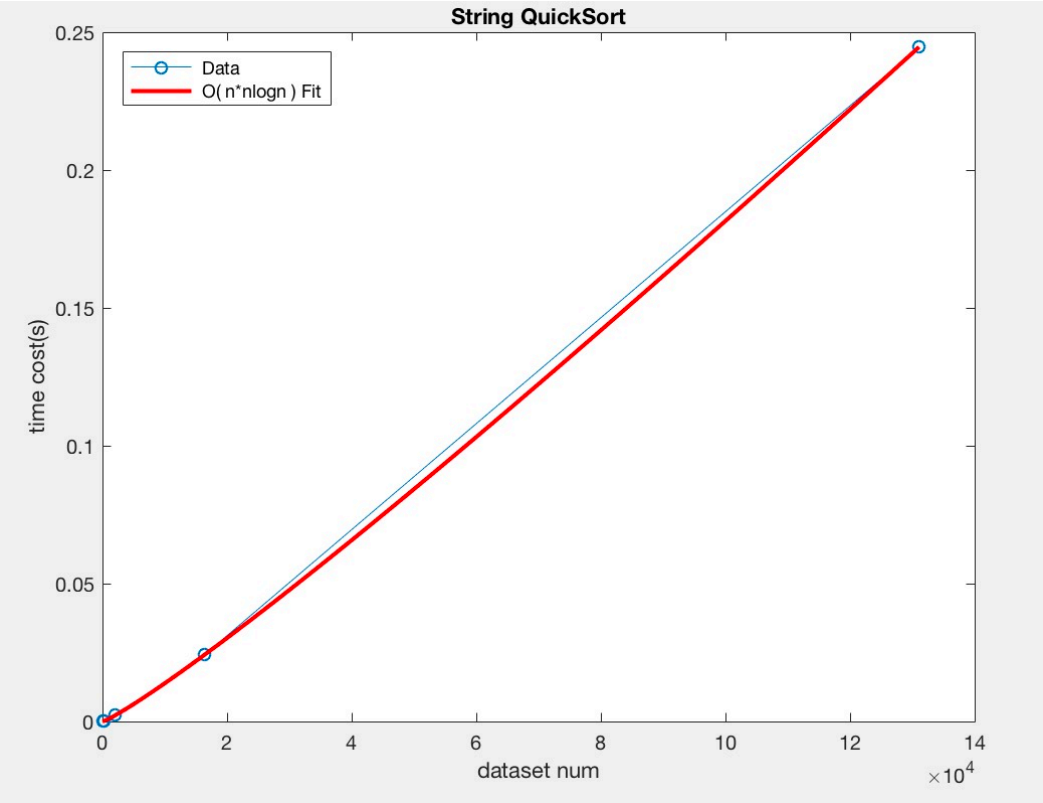
- 实验数据：（每组数据测试3次取平均，下表仅为平均结果，**原始数据请查看同级目录 orig_data.pdf** ）
    - 字符串排序（表中数据为时间 (s)）

	2^17	**2^14**	**2^11**	**2^8**	**2^5**	**2^2**
插入排序	147.071	2.38681	0.052307	0.0011539	0.00019446	8.657E-06
堆排序	0.282353	0.0319217	0.00271081	0.0003140	6.5493E-05	2.03E-06
归并排序	0.281993	0.0334950	0.00443534	0.0004881	6.8499E-05	7.925E-06
快速排序	0.244611	0.0241996	0.00226905	0.0002327	2.4551E-05	7.855E-06

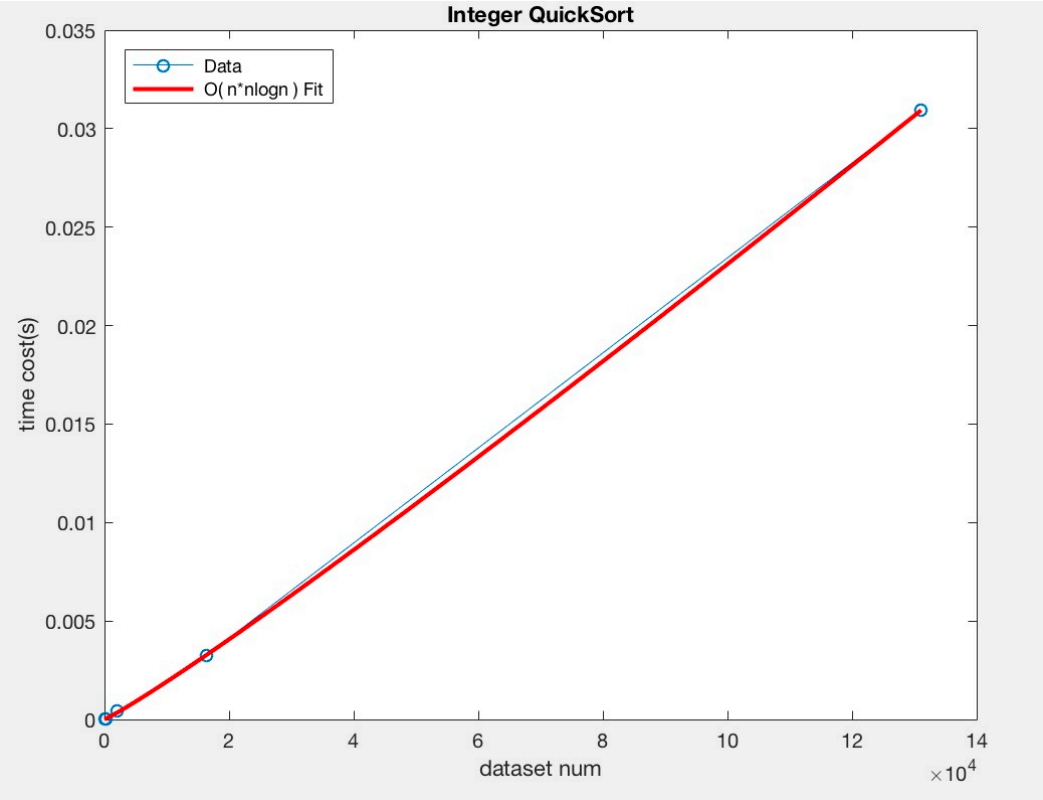
    - 整数排序（表中数据位时间 (s)）

	2^17	**2^14**	**2^11**	**2^8**	**2^5**	**2^2**
冒泡排序	76.3018	1.26572	0.0266077	0.0006131	6.3996E-05	1.309E-06
快速排序	0.03093	0.00326	0.0003965	4.2571E-05	6.28E-06	1.3E-06
基数排序	0.00522	0.00061	0.0002239	2.3164E-05	5.753E-06	3.616E-06
计数排序	0.00225	0.00064	0.0003671	0.00053667	0.000383112	0.0004674

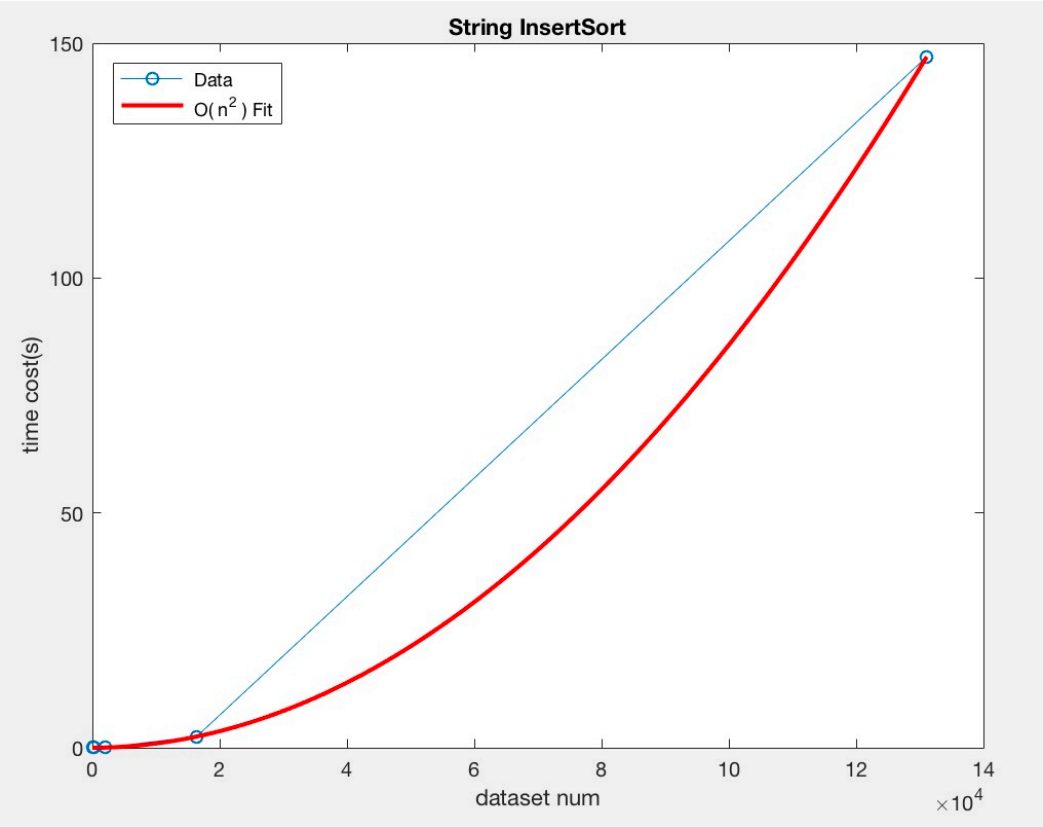
- 实验图表绘制
  - 采用 matlab 对相应算法采用相应函数拟合（如冒泡排序采用 O(n^(2)) 进行拟合，快速排序采用 O(nlogn) 进行拟合，拟合得到的曲线如下图所示：
    - 字符串快速排序：



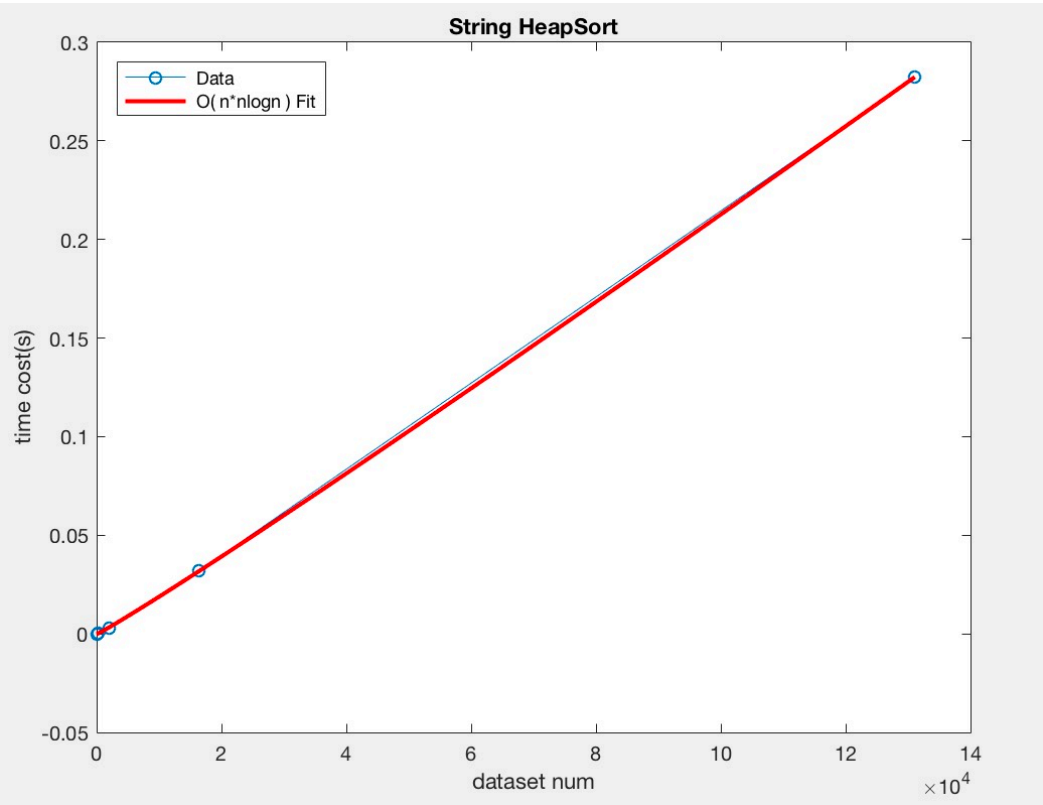
    - 整数快速排序:



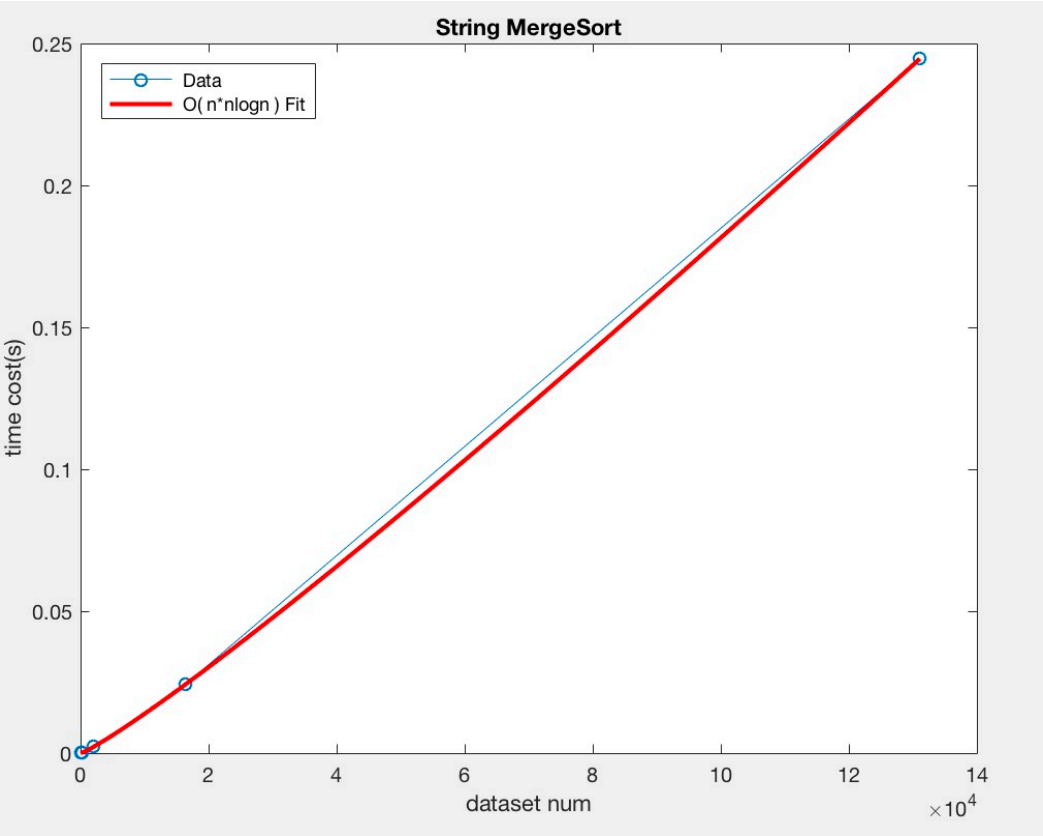
    - 字符串插入排序:



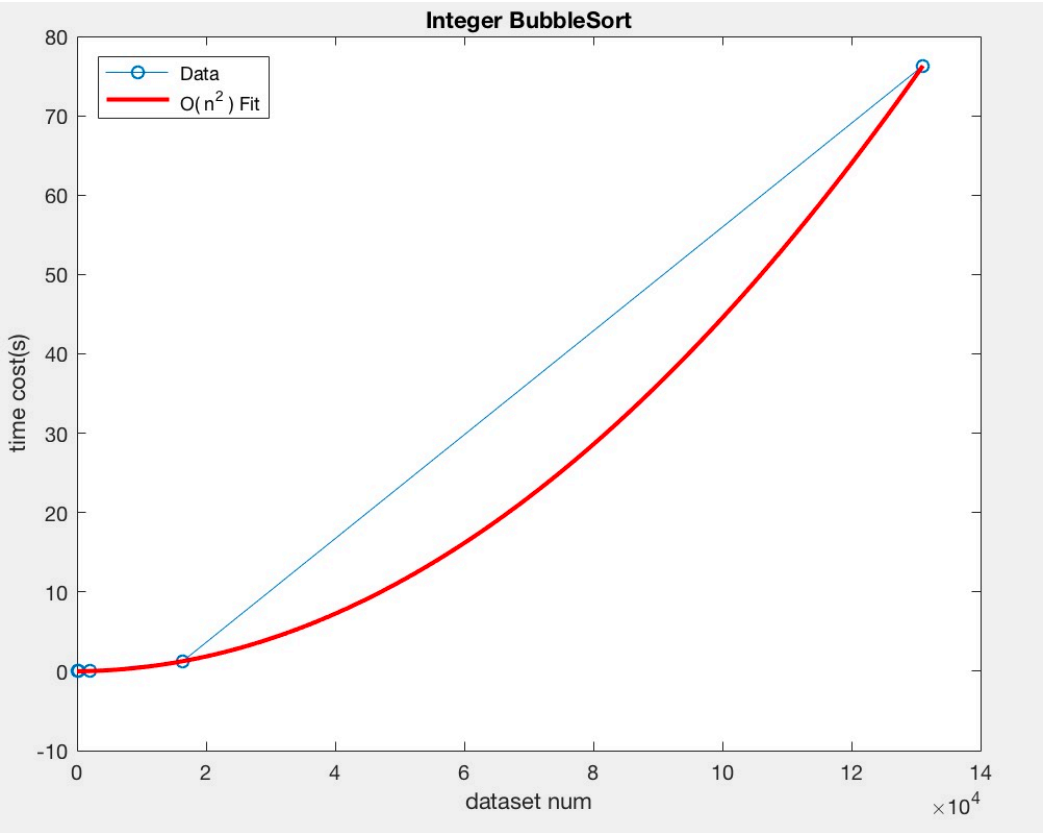
- 字符串堆排序:



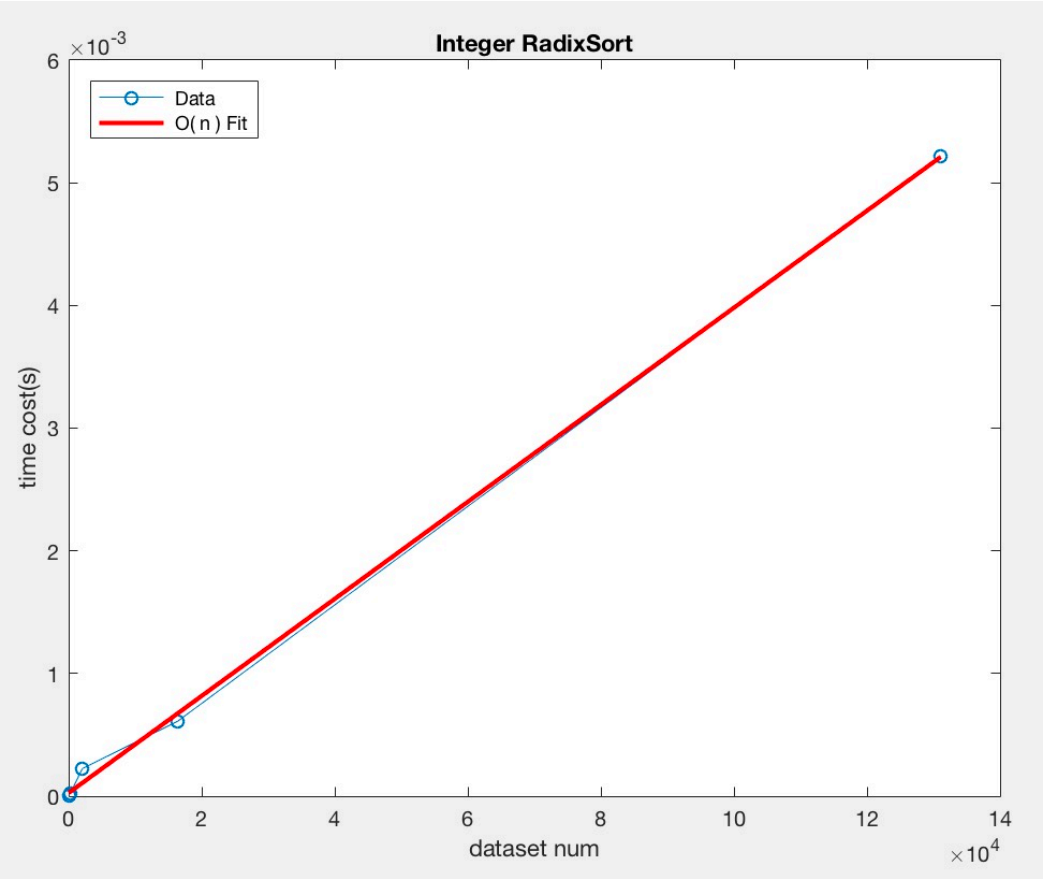
- 字符串归并排序:



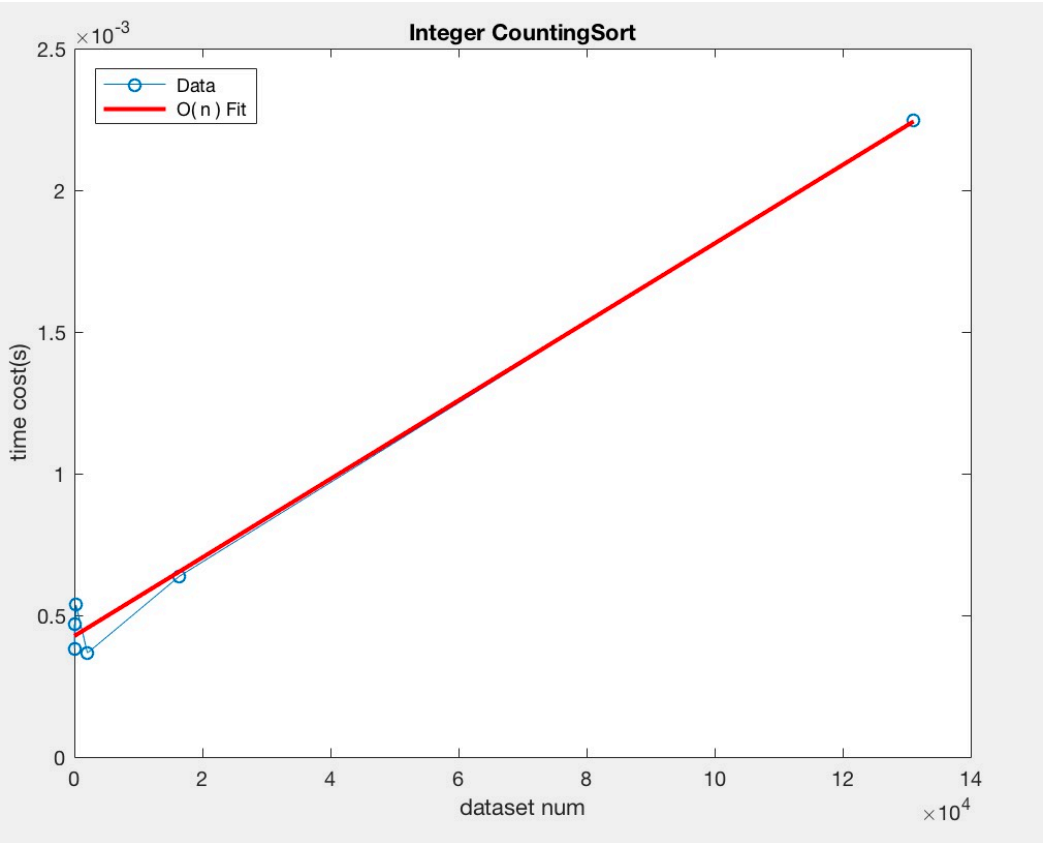
- 整数冒泡排序：



- 整数基数排序：



- 整数计数排序：



实验结果分析

- 横向比较：排序算法的优劣:

    - 从实验结果来看，快速排序在排序中体现了最好的性能，从理论上分析，快速排序的时间局部性和空间局部性都表现的非常良好，在 nlogn 渐进复杂度的算法中，归并排序表现的性能最差，其原因也是因为其空间局部性没有堆排序和归并排序好

- 纵向比较：算法的渐进复杂度：
    - 实际上，单从要求测试的数据分布很难体现出时间的渐进复杂度，原因有两点：
        - 数据规模过小：由于数据规模过小，大部分算法时间仅仅停留在1s 以内，甚至需要精确到微妙量级。如果程序在运行时系统有其他进程运行（如内核进程、子进程等），将会很大程度上影响算法的时间分析，从而对时间的渐进复杂度难以进行预测
        - 测试数据大小不合理：从实验结果图可以明显看出，按照指数系数做线性递减将会导致数据分布极其不均匀，对实验结果可能会造成很大影响。正确的测试数据量分布应当为线性递增的。
        - 随机数据无法准确预测渐进复杂度：由于算法的渐进复杂度与随机数据的分布相关（如已经顺序排序的随机数据可以在 O(n) 时间内排序完成），而当数据量增大时，新增的数据量的初始排序方式与未增加前的数据排序方式很难保持一致，因此所增加的排序时间很难保持一致，对预测造成一定困难。
    - 但从图中可以看出，大体上还是能满足原有理论对算法的时间渐进复杂度的估计(除计数排序以外)。
- 组间比较：对于不同的随机数据：
    - 不同的随机数据表现出来的算法执行时间可以差别很大，如冒泡排序在排2^17的数据量时，不同的随机数据排列可以导致一次随机数据执行的速度是另一次随机数据执行速度的好几倍（实验结果没有记录相关数据，但在实验过程中有所体会）。
- 异常情况分析:
    - 从最后一组计数排序的数据中可以看出，在数据规模较小的情况下，算法执行时间随数据规模的增加而减少。这种情况稳定出现，而出现这种情况的原因十分复杂，肯定涉及到了硬件层面有关 cache 和 页表等数据读取机制，其中有可能的原因是当数据量变大时，一次进入 cache 的数据量也会变大，从而使 cache miss 产生的概率减少(由于计数排序的三个循环都会造成大量的 cache miss)。从性能工具（如 Intel Vtune 或 gprof 等）的分析来看，第二个循环在数据量较小的时候会有非常大的起伏，由于第二个循环是（循环内）先访问 C[i] 后访问 C[i-1], 若不考虑循环整个过程则很大几率在访问 C[i - 1]时出现 cache miss，而考虑循环后 cache miss 出现的几率大大减少，但这也只是在 cache 足够大的情况下才可能出现。当 cache 有限时，不同的数据规模可能导致 cache 替换算法对 数组 C 在 cache 中的数据造成不同程度的影响 (操作系统课程中已经体现过)，即有可能在本次循环中存入 C[i] 而在下次循环开始时因为需要访问其他项而将 C[i]替换出去，造成下次循环的 cache miss，这样的行为将大幅降低计数排序的性能，导致最后结果出现的波动。

实验心得

1. 本实验重点在于体验了不同排序算法间的差异以及对排序速度有了一定量化上的认识。
2. 个人认为本实验对于要求测试数据量分布并不合理，但由于时间有限，以及自身以前有过算法比较经历，所以在本实验中不再展开。
3. 本实验可以写一个脚本针对数据量进行大规模测试，经 C++输入进.dat 文件后再由 matlab 脚本进行自动拟合来得到数据规模与算法执行时间的关系。

代码编译及使用

- 操作系统要求: Linux 或者 MacOS
- 编译器要求：g++ （能支持 C++11语法）
- 编译 (ex1, ex2分别只能执行 `$./sort str` 和 `$./sort int`)

```
$ cd into src directory
$make
for generating new random dataset (there already exists datasets in
input/ directory)
$./gen
for sorting string:
$./sort str
for sorting integers:
$./sort int
```