实验 1　Sorting

1. 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：2^2，2^5，2^8，2^{11}，2^{14}，2^{17}；

  算法：直接插入排序，堆排序，归并排序，快速排序。
  - 字符串大小判断标准：
    - 1.首先按字符串长度进行排序（短字符串在前，长字符串在后）。
    - 2.对长度相同的字符串，按字母顺序进行排序。

- 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：　2^2，2^5，2^8，2^{11}，2^{14}，2^{17}　；

  算法：冒泡排序，快速排序，　基数排序，计数排序。

2. 实验环境

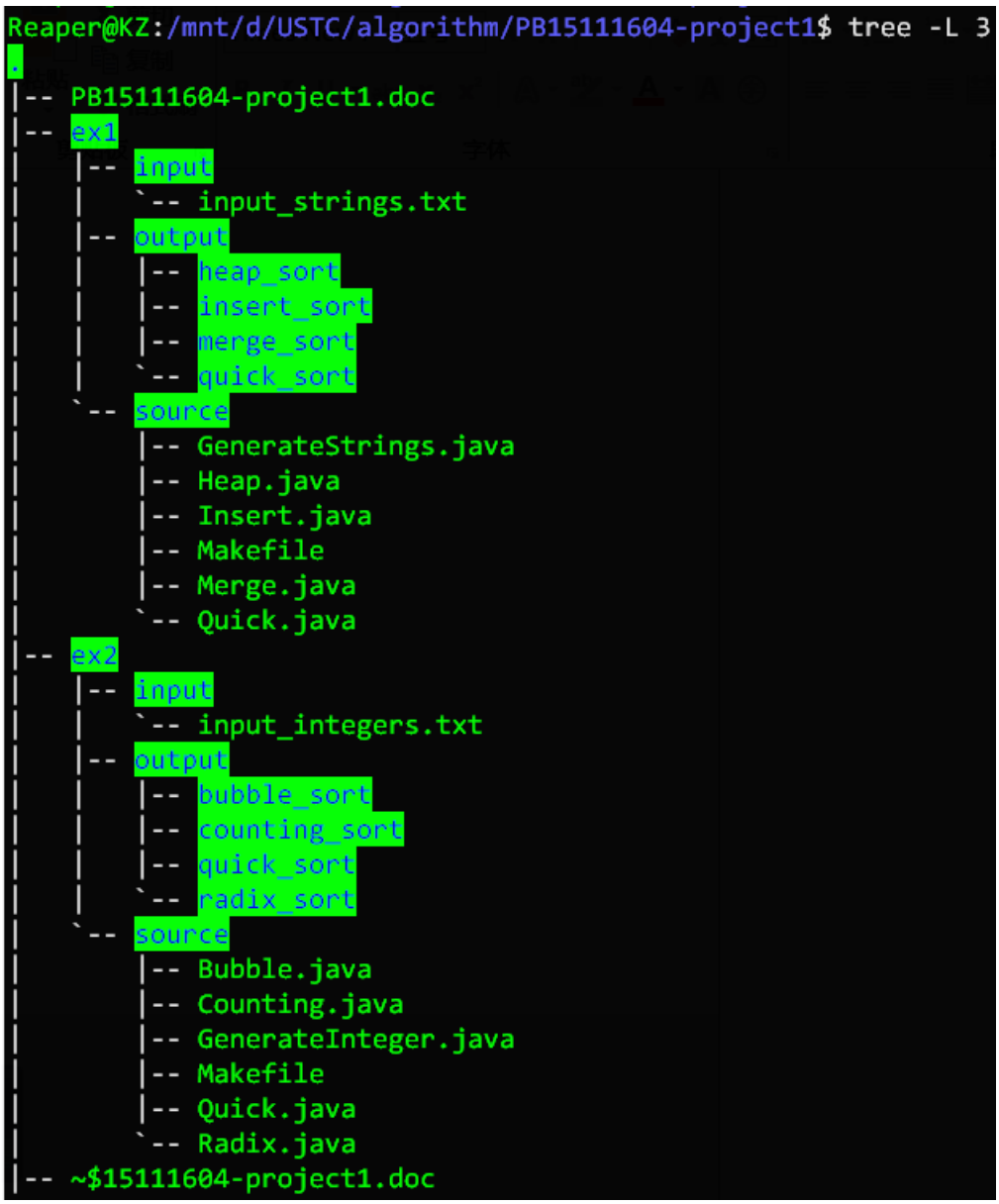
编译环境：Ubuntu 14.04 (WSL- Windows Subsystem for Linux)
　　OpenJDK version "1.8.0_144"
机器内存：16G
时钟主频：2.4GHz

3. build

我写了两个 Makefile，分别位于 source 文件夹中，以 ex1 中为例，如下：

```
all: insert heap merge quick

generate:
    javac GenerateStrings.java;
    java GenerateStrings;

insert:
    javac -encoding utf-8 Insert.java;
    java Insert;

heap:
    javac -encoding utf-8 Heap.java;
    java Heap;

merge:
    javac -encoding utf-8 Merge.java;
    java Merge;

quick:
    javac -encoding utf-8 Quick.java;
    java Quick;

clean:
    rm *.class
```

最后的**目录结构**：



result_n.txt 都在 output/*_sort/中。

4. 实验过程

1. 生成随机数与随机字符串
   a) 随机数：
      i. 生成 (1<<17) 个 (int)(Math.random()*65535 + 1) 到 input_integers.txt 中。
      ii. 由于刚学 java，难免会遇到“奇怪的”问题，比如 println 到文件，但输出的数字个数总是比生成的要少，经过层层推理追踪，最终发现是缓冲区没有 flush。学到新东西！
   b) 随机字符串：
      i. 先生成随机的字符串长度 1-32，再随机生成每个字符串的字符，两层循环。
2. ex1：

a) 直接插入：
   i. 最简单。作为 ex1 的第一个算法，主要的任务就是为后面的算法设置一个可以套用的框架，以方便模块化。
   ii. 为了测量时间，用的是 System.nanoTime()函数，但由于返回的最后 3 位都是 0，所以取微妙为单位。
   iii. 为了与教材一致，统计时间时是从形成一个用于排序的字符串开始，到最终生成字符串结束计时。其他磁盘 io 的时间没有考虑，因为教材上的例子都是直接从内存中现有的数组开始算的，没有考虑磁盘 io。
b) 堆排序：
   i. 大致思路与教材相同，单独需要考虑的是数组是从 0 开始而非 1，所以例如 left(),right(),parent()之类的函数以及其他函数的细节需要调整。
c) 快速排序：
   i. 大致思路与教材相同
d) 归并排序：
   i. 大致思路与教材相同，只不过教材里的“无穷大”，需要在这里改为比如 33 个’A’构成的字符串来表示。

3. ex2:
   a) 快速排序：
      i. 与 ex1 的快速排序类似，所以作为 ex2 的第一个算法，主要的任务就是为 ex2 后面的算法提供一个框架。由于 ex2 是比较 int，所以将 ex1 的比较函数 compare 去掉，改成 int 的直接比较。还有其他关于 int 类型细节的调整。
   b) 冒泡排序：
      i. 可以说是最简单的了。
   c) 计数排序：
      i. 由于基数排序可以用到计数排序，所以先写计数排序。
      ii. 由于计数排序不是就地排序，所以需要在函数外单独设置一个用于存储排好序的数组的 B。
      iii. 不同于教材，我把 C 数组设置为了 0-35534，一个原因是为了节省空间（虽然只有一个 int），还有一个原因是为了加深对计数排序的理解，毕竟之前没有接触过。
      iv. 实验过程中发现随着 n 的数量级的增加，T 并没有按照想象中的递增关系，而是在 n 比较小的时候，比如 2,5,8 会有较大波动，到 11 后面比较正常。所以我为了便于分析，多生成了随机数据，总共 2^23 个数据。并且增加了 2^18,19,20,21,22,23 的排序，发现结果较为理想。
   d) 基数排序：
      i. 基数排序写在计数排序后面，是为了方便写完计数之后移植。
      ii. 我选取的基数是 16，也就是每 4bit 算作一位，最大值为 65535，所以只需要 4 次即可。
      iii. 需要注意的是，从计数排序移植的时候，不仅要改掉 65535，而且要注意之前的取值范围是 1-65535，这次是 0-15，所以要注意

C 的 index 要调整。实验过程中正是因为这一点让我对基数排序的理解进一步加深。

iv. 另外要注意的是，由于计数排序不是就地排序，所以每执行一次之后需要将新的 B 数组赋值给用来排序的数组 A。

4. 制图

a) 利用 excel，origin 处理，得到曲线图。

5. **实验关键代码截图（结合文字说明）**

根据前面**实验过程**提到的关键部分进行说明。

1. ex1：
   a) 直接插入（ex1 框架说明）：
      i. 最简单。作为 ex1 的第一个算法，主要的任务就是为后面的算法设置一个可以套用的框架，以方便模块化。

整体框架：

```
/*************************************************************
**********
    > File Name: Insert.java
    > Author: King.Zevin
    > Student Number: PB15111604
    > Mail: jzw0222@mail.ustc.edu.cn
    > QQ: 1033329461

 *************************************************************
 *********/

import java.util.*;
import java.nio.file.*;
import java.io.*;

public class Insert{
    public static void main(String[] args) throws IOException{

    }

    // 用于比较字符串大小的函数
    // 小则-1 大则1 相等则0
    public static int compare(String a, String b){
    }

    // 排序函数，返回纳秒数，便于计时。
    public static long sort(String[] A){
    }
}
```

首先 import 需要的 packages，

其次有一个 main 主体函数，一个用于比较字符串大小的 compare 函数，一个用于排序，并且返回纳秒数的 sort 函数。

先说 compare：

根据既定规则，代码如下：

```
46
47      // 用于比较字符串大小的函数
48      // 小则-1 大则1 相等则0
49      public static int compare(String a, String b){
50          if(a.length() < b.length())
51              return -1;
52          else if(a.length() > b.length())
53              return 1;
54          else
55              return a.compareTo(b);
56      }
57
```

再说 main：

```
15      public static void main(String[] args) throws IOException{
16          /* 初始化:读入所有数据到origin数组中.*/
17          Scanner in = new Scanner(Paths.get("../input/
                input_strings.txt"), "UTF-8");
18          PrintWriter outTime = new PrintWriter("../output/
                insert_sort/time.txt", "UTF-8");
19          String[] origin=new String[1<<17];
20          for(int i = 0; i < 1<<17; i++){
21              origin[i] = in.nextLine();
22          }
23
24          int[] exp={2, 5, 8, 11, 14, 17};
25          // 开始调用函数进行排序.
26          for(int index : exp){
27              String[] partArray = Arrays.copyOf(origin, 1<<index)
                    ;
28              // 取微秒级, 因为发现纳秒后三位都是0.
29              long enduration = sort(partArray)/1000;
30
31              // 输出时间到time.txt, 注意flush().
32              outTime.println("index: " + index + "\ntime: " +
                    enduration + "\tmicroseconds.");
33              outTime.flush();
34
35              // 输出排好序的数组到result中
36              System.out.println("index: " + index + "\ntime: " +
                    enduration + "\tmicroseconds.");
37              PrintWriter outSort = new PrintWriter("../output/
                    insert_sort/result_"+index+".txt", "UTF-8");
38              for(int j = 0; j < 1<<index; j++){
39                  outSort.println(partArray[j]);
40                  outSort.flush();
41              }
42          }
43      }
```

首先读入所有的 2^17 个字符串，按照 2,5,8,11,14,17 的顺序调用 sort 进行排序计时，之后输出到规定文件中。

最后说 sort：

```
57      public static Long sort(String[] A){
58          // 开始计时
59          Long startTime = System.nanoTime();
60          // 排序
61          // 算法之间主要只有这里不同。
62          // 初始化
63          int l = A.length;
64          int min;
65          String tmp;
66
67          for(int i = 0; i < l - 1; i++){
68              min = i;
69              for(int j = i+1 ; j < l; j++){
70                  if(compare(A[min], A[j]) > 0){
71                      min = j;
72                  }
73              }
74              // 交换
75              tmp = A[min];
76              A[min] = A[i];
77              A[i] = tmp;
78          }
79
80          // 结束计时，返回时间差。
81          Long endTime = System.nanoTime();
82          return endTime - startTime;
83      }
```

为了测量时间，用的是 System.nanoTime()函数，但由于返回的最后 3 位都是 0，所以最终取微妙为单位。

各个算法之间主要的差异就在中间开始计时到结束计时的代码部分。对于插入排序，算法如上，很简单。

另外，为了与教材一致，统计时间时是从形成一个用于排序的字符串开始，到最终生成字符串结束计时。其他磁盘 io 的时间没有考虑，因为教材上的例子都是直接从内存中现有的数组开始算的，没有考虑磁盘 io。

b) 堆排序：

i. 大致思路与教材相同，单独需要考虑的是数组是从 0 开始而非 1，所以例如 left(),right(),parent()之类的函数以及其他函数的细节需要调整。

```
77      public static int left(int i){
78          return 2 * i + 1;
79      }
80
81      public static int right(int i){
82          return 2 * i + 2;
83      }
84
85      public static int parent(int i){
86          return (i + 1) / 2 - 1;
87      }
```

ii. 另外需要注意的是 heapSize 需要设置为函数外变量

```
14  public class Heap{
15
16      // 为了避免单独为了heapSize弄个class,
17      // 所以设置为函数外变量
18      public static int heapSize = 0;
19
```

```
112     public static void heap_sort(String[] A){
113         String tmp;
114
115         heapSize = A.length;
116         build_max_heap(A);
117
118         for(int i = A.length - 1; i >= 1; i--){
119             // 交换A[0],A[i]
120             tmp = A[i];
121             A[i] = A[0];
122             A[0] = tmp;
123             heapSize --;
124             max_heapify(A, 0);
125         }
126     }
127 }
```

c) 快速排序：

i. 大致思路与教材相同

调用：

```
// 排序函数，返回纳秒数，便于计时。
public static long sort(String[] A){
    // 开始计时
    long startTime = System.nanoTime();
    // 排序
    // 算法之间主要只有这里不同。
    quick_sort(A, 0, A.length - 1);

    // 结束计时，返回时间差。
    long endTime = System.nanoTime();
    return endTime - startTime;
}
```

```
91
92      public static void quick_sort(String[] A, int p, int r){
93          if(p < r){
94              int q = partition(A, p, r);
95              quick_sort(A, p, q-1);
96              quick_sort(A, q+1, r);
97          }
98      }
99  }
```

Partition 函数：

```
71      // partition
72      public static int partition(String[] A, int p, int r){
73          String x = A[r];
74          String tmp;
75          int i = p - 1;
76          for(int j = p; j < r; j++){
77              if(compare(A[j], x) <= 0){
78                  i++;
79                  // exchange A[i], A[j]
80                  // 确保A[i]是小于等于x的.
81                  tmp = A[j];
82                  A[j] = A[i];
83                  A[i] = tmp;
84              }
85          }
86          i++;
87          A[r] = A[i];
88          A[i] = x;
89          return i;
90      }
```

d) 归并排序：

i. 大致思路与教材相同，只不过教材里的“无穷大”，需要在这里改为比如 33 个’A’构成的字符串来表示。

调用：

```
59
60          // 排序函数，返回纳秒数，便于计时。
61          public static long sort(String[] A){
62              // 开始计时
63              long startTime = System.nanoTime();
64              // 排序
65              // 算法之间主要只有这里不同。
66              merge_sort(A, 0, A.length - 1);
67
68              // 结束计时，返回时间差。
69              long endTime = System.nanoTime();
70              return endTime - startTime;
71          }
```

```
102
103     public static void merge_sort(String[] A, int p, int r){
104         if(p < r){
105             int q = (p+r) / 2;
106             merge_sort(A, p, q);
107             merge_sort(A, q+1, r);
108             merge(A, p, q, r);
109         }
110     }
111 }
112
```

merge：

```
73      // merge
74      public static void merge(String[] A, int p, int q, int r){
75          int n1 = q - p + 1;
76          int n2 = r - q;
77          String[] L = new String[n1 + 1];
78          String[] R = new String[n2 + 1];
79          for(int i = 0; i < n1; i++){
80              L[i] = A[p + i];
81          }
82          for(int i = 0; i < n2; i++){
83              R[i] = A[q + i + 1];
84          }
85          // 33个'A'字符串，表示最大
86          L[n1] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
87          R[n2] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
88
89          int m = 0;
90          int n = 0;
91          for(int k = p; k <= r; k++){
92              if(compare(L[m], R[n]) <= 0){
93                  A[k] = L[m];
94                  m++;
95              }
96              else{
97                  A[k] = R[n];
98                  n++;
99              }
100         }
101     }
102
```

2. ex2:
    a) 快速排序（ex2 框架说明）：
        i. 与 ex1 的快速排序类似，所以作为 ex2 的第一个算法，主要的任务就是为 ex2 后面的算法提供一个框架。由于 ex2 是比较 int，所以将 ex1 的比较函数 compare 去掉，改成 int 的直接比较。还有其他关于 int 类型细节的调整。

```
/*************************************************************
******
    > File Name: Quick.java
    > Author: King.Zevin
    > Student Number: PB15111604
    > Mail: jzw0222@mail.ustc.edu.cn
    > QQ: 1033329461

 *************************************************************
 *****/

import java.util.*;
import java.nio.file.*;
import java.io.*;

public class Quick{

    public static void main(String[] args) throws IOException{
    }

    // 排序函数，返回纳秒数，便于计时。
    public static long sort(int[] A){
    }

    // partition
    public static int partition(int[] A, int p, int r){
    }

    public static void quick_sort(int[] A, int p, int r){
    }
}
```

可以看出，没有 compare 函数了。

b) 冒泡排序：

```
60      public static void bubble_sort(int[] A){
61          int tmp;
62          int l = A.length;
63          for(int i = 0; i < l - 1; i++){
64              for(int j = 1; j < l; j++){
65                  if(A[j - 1] > A[j]){
66                      // 交换A[j], A[j - 1]
67                      tmp = A[j];
68                      A[j] = A[j - 1];
69                      A[j - 1] = tmp;
70                  }
71              }
72          }
73      }
74  }
```

i. 可以说是最简单的了。

c) 计数排序：

i. 由于基数排序可以用到计数排序，所以先写计数排序。

ii. 由于计数排序不是就地排序，所以需要在函数外单独设置一个用于存储排好序的数组的 B。

```
14  public class Counting{
15
16      // 由于计数排序不是就地排序，
17      // 所以需要在函数外，单独设置
18      // 一个用于存储排好序的数组的B。
19      public static int[] B;
```

iii. 不同于教材，我把 C 数组设置为了 0-35534，一个原因是为了节省空间（虽然只有一个 int），还有一个原因是为了加深对计数排序的理解，毕竟之前没有接触过。

```
71      public static void counting_sort(int[] A){
72          int tmp;
73          int l = A.length;
74          B = new int[l];
75          int[] C = new int[65535];
76
77          // s = System.nanoTime();
78          for(int i = 0; i < l; i++){
79              C[A[i]-1]++;
80          }
81          // C[i] 的值即为大小为i-1的值的个数
82
83          for(int i = 1; i < 65535; i++){
84              C[i] += C[i-1];
85          }
86          // C[i] 的值即为大小为小于等于i-1的值的个数
87
88          for(int i = l-1; i >= 0; i--){
89              B[C[A[i]-1]-1] = A[i];
90              C[A[i]-1] --;
91          }
92      }
93  }
```

iv. 实验过程中发现随着 n 的数量级的增加，T 并没有按照想象中的递增关系，而是在 n 比较小的时候，比如 2,5,8 会有较大波动，到 11 后面比较正常。所以我为了便于分析，多生成了随机数据，总共 2^23 个数据。并且增加了 2^18,19,20,21,22,23 的排序，发现结果较为理想。

```
29
30          // int[] exp={17, 14, 11, 8, 5, 2};
31          int[] exp={2, 5, 8, 11, 14, 17, 18, 19, 20, 21, 22, 23};
```

d) 基数排序：

i. 我选取的基数是 16，也就是每 4bit 算作一位，最大值为 65535，所以只需要 4 次即可。

调用：

```
55      // 排序函数，返回纳秒数，便于计时。
56      public static Long sort(int[] A){
57          // 开始计时
58          Long startTime = System.nanoTime();
59          // 排序
60          // 算法之间主要只有这里不同。
61          radix_sort(A, 4);
62
63          // 结束计时，返回时间差。
64          Long endTime = System.nanoTime();
65          return endTime - startTime;
66      }
```

digit：

```
75      // 返回第d位数(16进制),规定低位为第1位.
76      // 如0xef的第1位为0xf,第2位为0xe.
77      public static int digit(int num, int d){
78          // System.out.printf("%x : %d", num, d);
79          // System.out.printf("\t%x\n", (num >> (4*(d-1))) & 0x0f);
80          return (num >> (4*(d-1))) & 0x0f;
81      }
```

ii. 需要注意的是，从计数排序移植的时候，不仅要改掉 65535，而且要注意之前的取值范围是 1-65535，这次是 0-15，所以要注意 C 的 index 要调整。实验过程中正是因为这一点让我对基数排序的理解进一步加深。

调整：

```
83      public static void counting_sort(int[] A, int d){
84          int tmp;
85          int l = A.length;
86          B = new int[l];
87          int[] C = new int[16];
88
89          for(int i = 0; i < l; i++){
90              C[digit(A[i], d)]++;
91          }
92          // C[i] 的值即为大小为i的值的个数
93
94          for(int i = 1; i < 16; i++){
95              C[i] += C[i-1];
96          }
97          // C[i] 的值即为大小为小于等于i的值的个数
98
99          for(int i = l-1; i >= 0; i--){
100             B[C[digit(A[i], d)]-1] = A[i];
101             C[digit(A[i], d)] --;
102         }
103     }
104 }
```

iii. 另外要注意的是，由于计数排序不是就地排序，所以每执行一次之后需要将新的 B 数组赋值给用来排序的数组 A。

```
68      public static void radix_sort(int[] A, int d){
69          for(int i = 1; i <= d; i++){
70              counting_sort(A, i);
71              A = B;
72          }
73      }
```

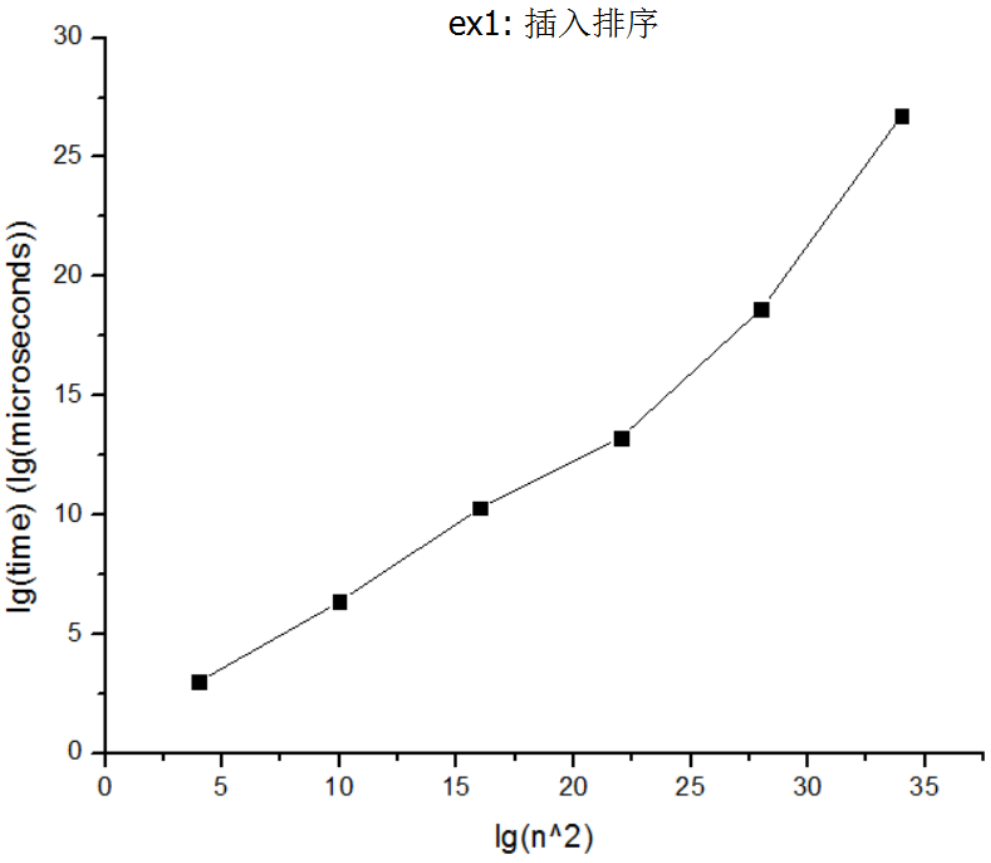
6. 实验结果、分析（结合相关数据图表分析）

a) ex1:

i. 插入排序：

按照 time.txt 得到：

ex1:插入排序				
n	n^2	time(microseconds)	lg(n^2)	lg(time)
4	16	8	4	3
32	1,024	81	10	6.33985
256	65,536	1,239	16	10.275
2,048	4,194,304	9,567	22	13.2239
16,384	268,435,456	402,840	28	18.6198
131,072	17,179,869,184	111,579,000	34	26.7335



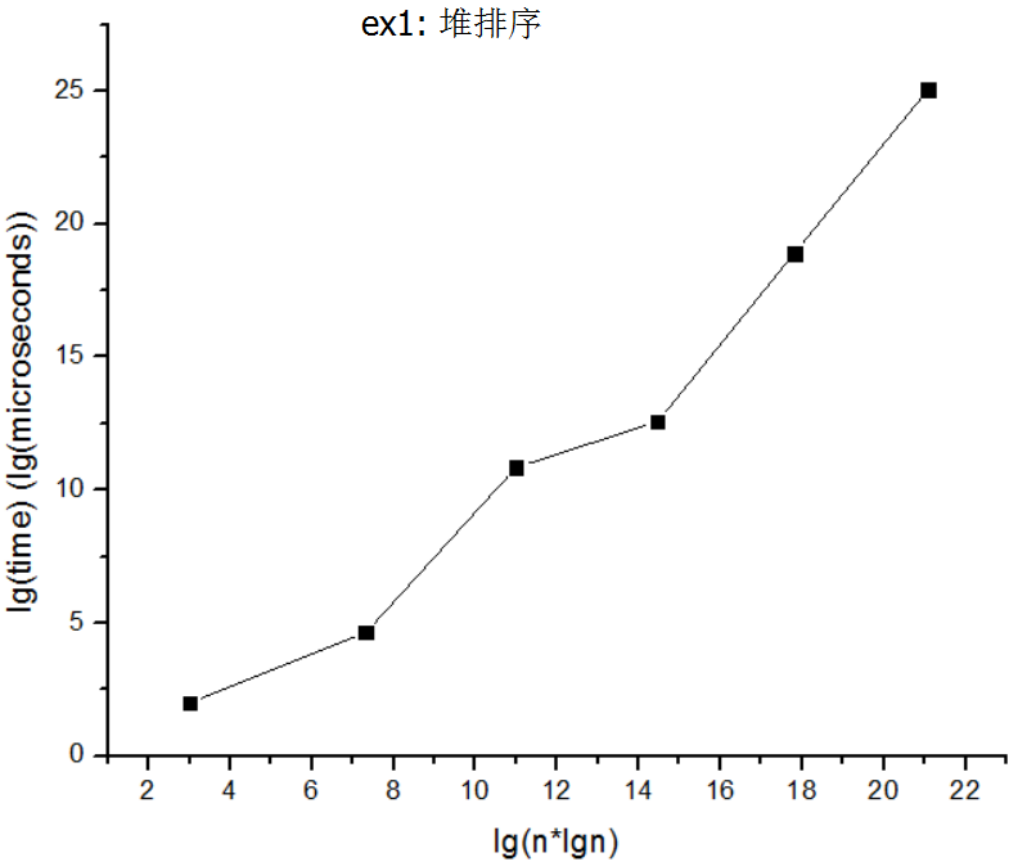
结果较为符合 $\Theta(n^2)$ 的期望。

ii. 堆排序：

得到

ex1:堆排序				
n	n*lgn	time(microseconds)	lg(n*lgn)	lg(time)
4	8	4	3	2
32	160	25	7.3219281	4.64386
256	2,048	1,837	11	10.8431
2,048	22,528	6,049	14.459432	12.5625
16,384	229,376	488,874	17.807355	18.8991
131,072	2,228,224	34,780,567	21.087463	25.0518

做图得

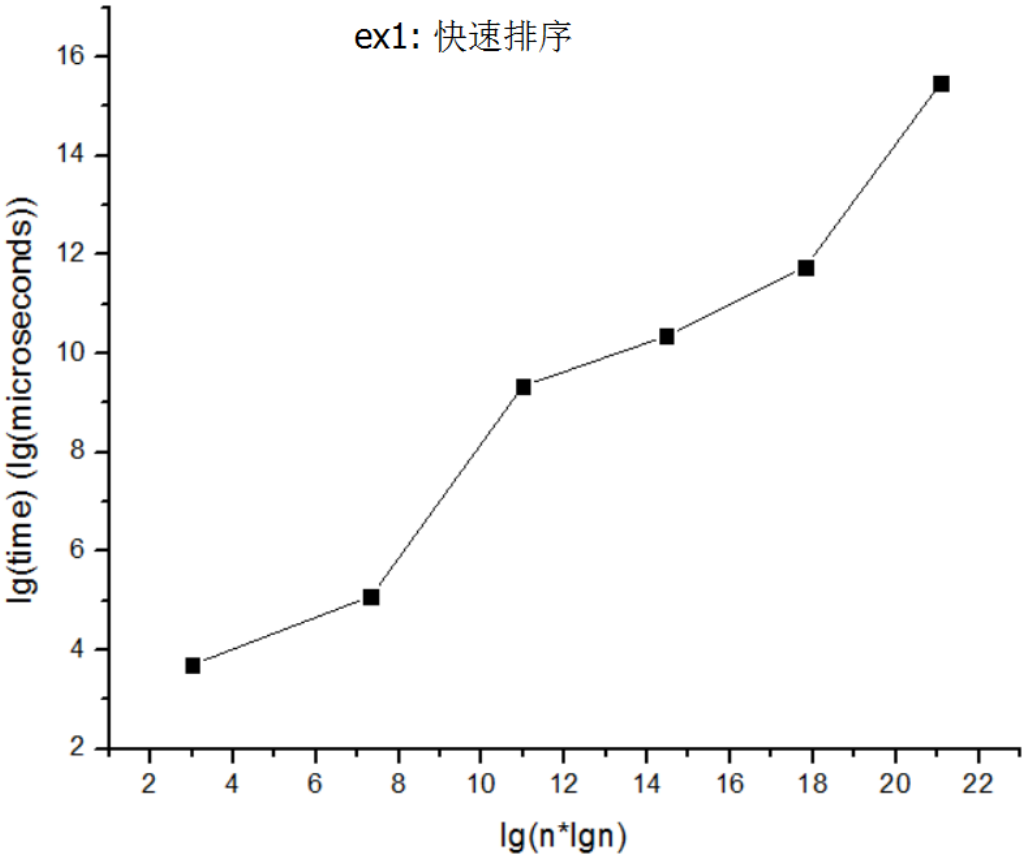


结果较为符合 $\Theta(n \lg n)$ 的期望。

iii. 快速排序：

得到

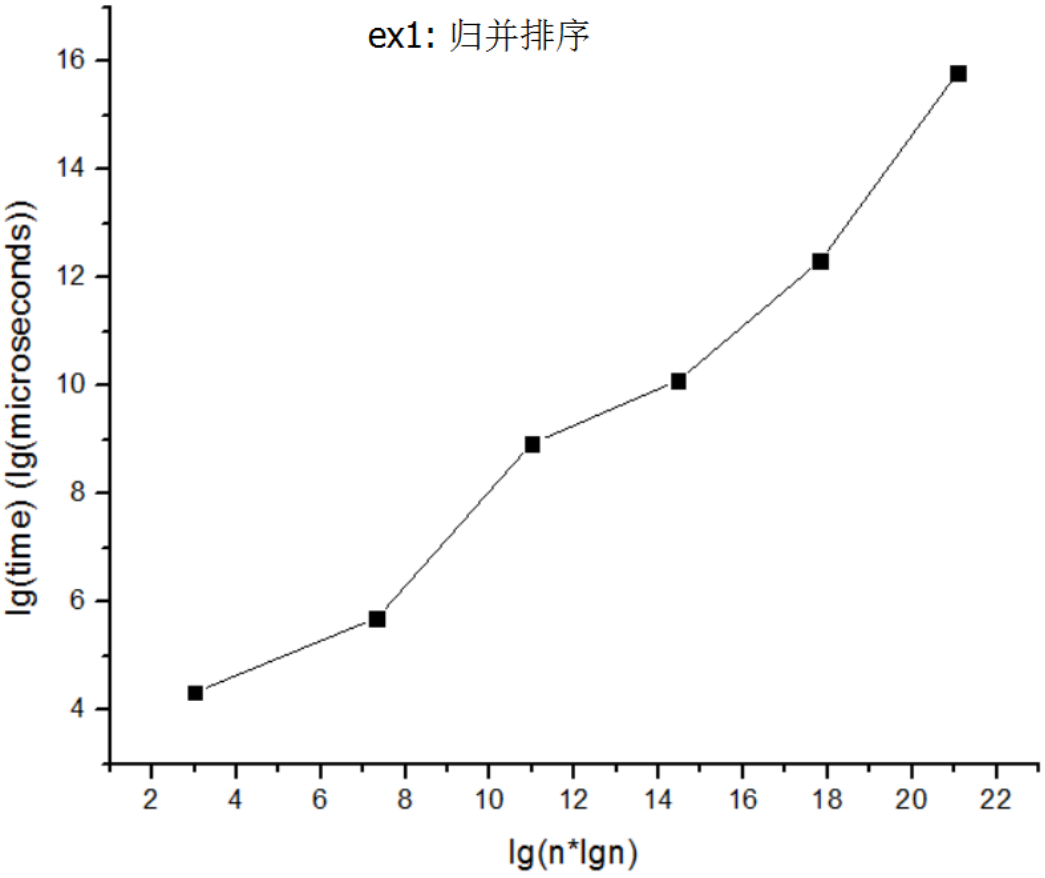
ex1:快速排序				
n	n*lgn	time(microseconds)	lg(n*lgn)	lg(time)
4	8	13	3	3.70044
32	160	34	7.3219281	5.08746
256	2,048	648	11	9.33985
2,048	22,528	1,311	14.459432	10.3565
16,384	229,376	3,441	17.807355	11.7486
131,072	2,228,224	45,305	21.087463	15.4674



结果较为符合 $\Theta(n \lg n)$ 的期望。

iv. 归并排序：

ex1:归并排序				
n	n*lgn	time(microseconds)	lg(n*lgn)	lg(time)
4	8	20	3	4.32193
32	160	52	7.3219281	5.70044
256	2,048	487	11	8.92778
2,048	22,528	1,089	14.459432	10.0888
16,384	229,376	5,018	17.807355	12.2929
131,072	2,228,224	56,801	21.087463	15.7936

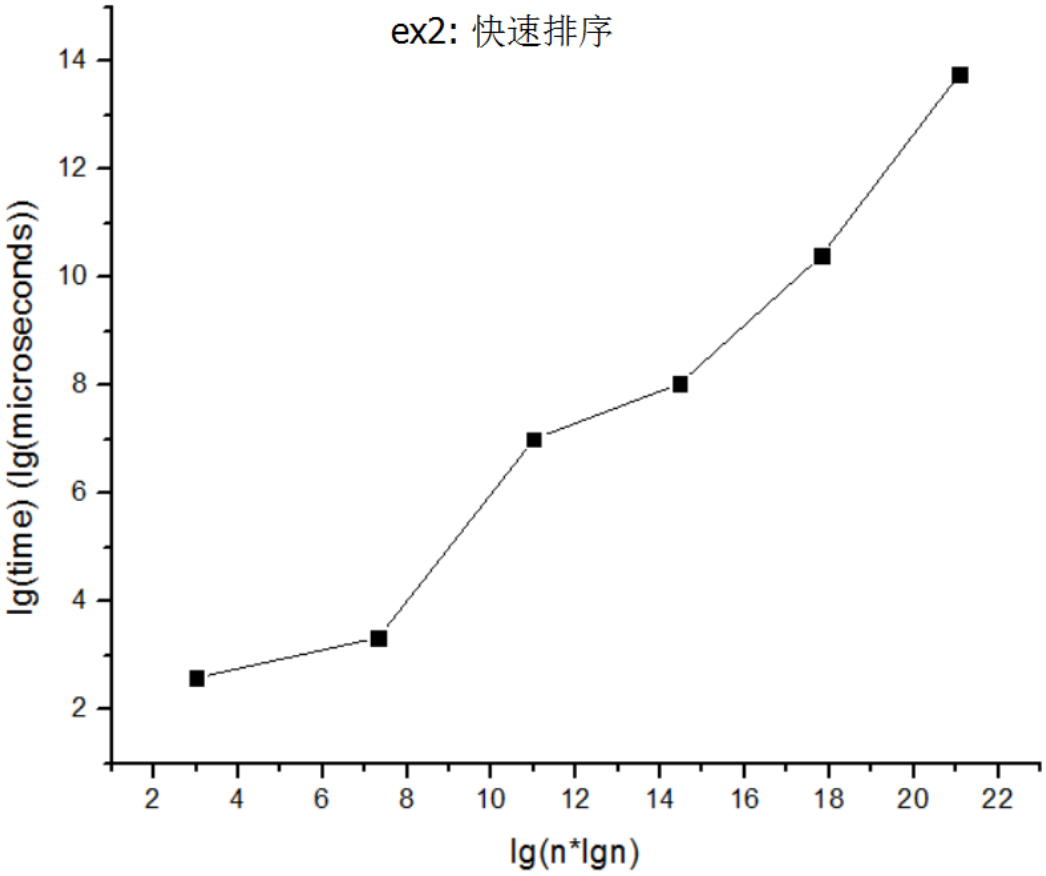


结果较为符合 $\Theta(n \lg n)$ 的期望。

b) ex2:

i. 快速排序：

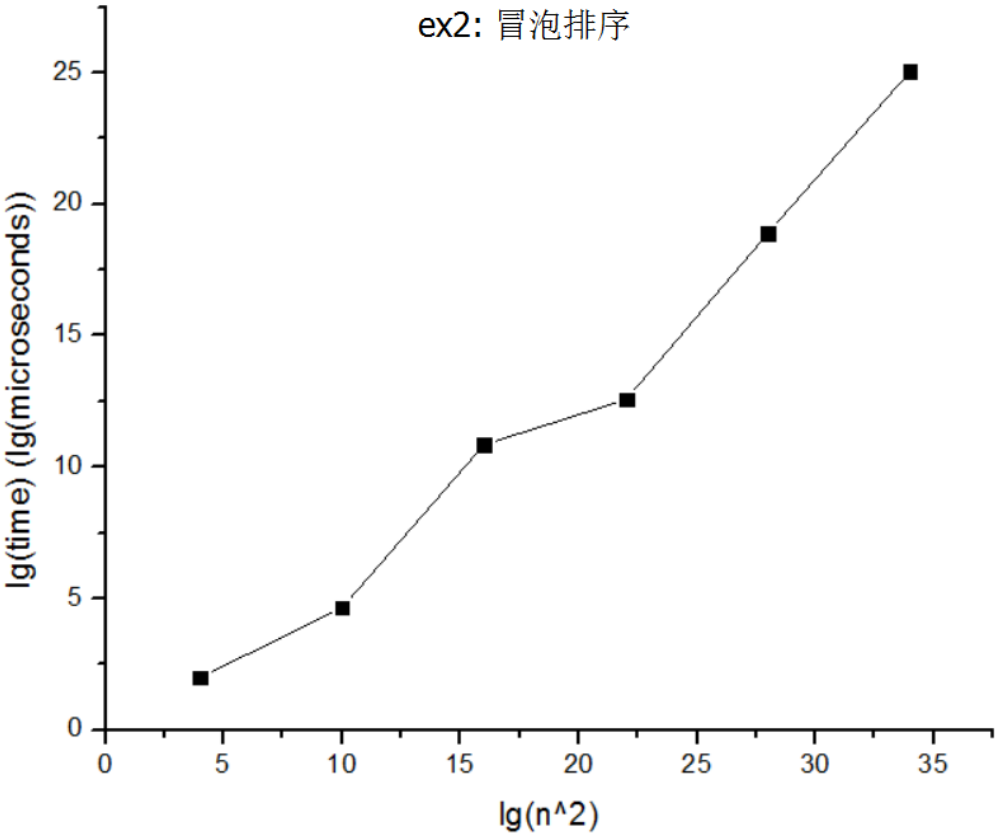
ex2:快速排序				
n	n*lgn	time(microseconds)	lg(n*lgn)	lg(time)
4	8	6	3	2.58496
32	160	10	7.3219281	3.32193
256	2,048	128	11	7
2,048	22,528	262	14.459432	8.03342
16,384	229,376	1,344	17.807355	10.3923
131,072	2,228,224	13,876	21.087463	13.7603



结果较为符合 $\Theta(n \lg n)$ 的期望。

ii. 冒泡排序：

ex2:冒泡排序				
c	n^2	time(microseconds)	lg(n^2)	lg(time)
4	16	4	4	2
32	1,024	25	10	4.64386
256	65,536	1,837	16	10.8431
2,048	4,194,304	6,049	22	12.5625
16,384	268,435,456	488,874	28	18.8991
131,072	17,179,869,184	34,780,567	34	25.0518



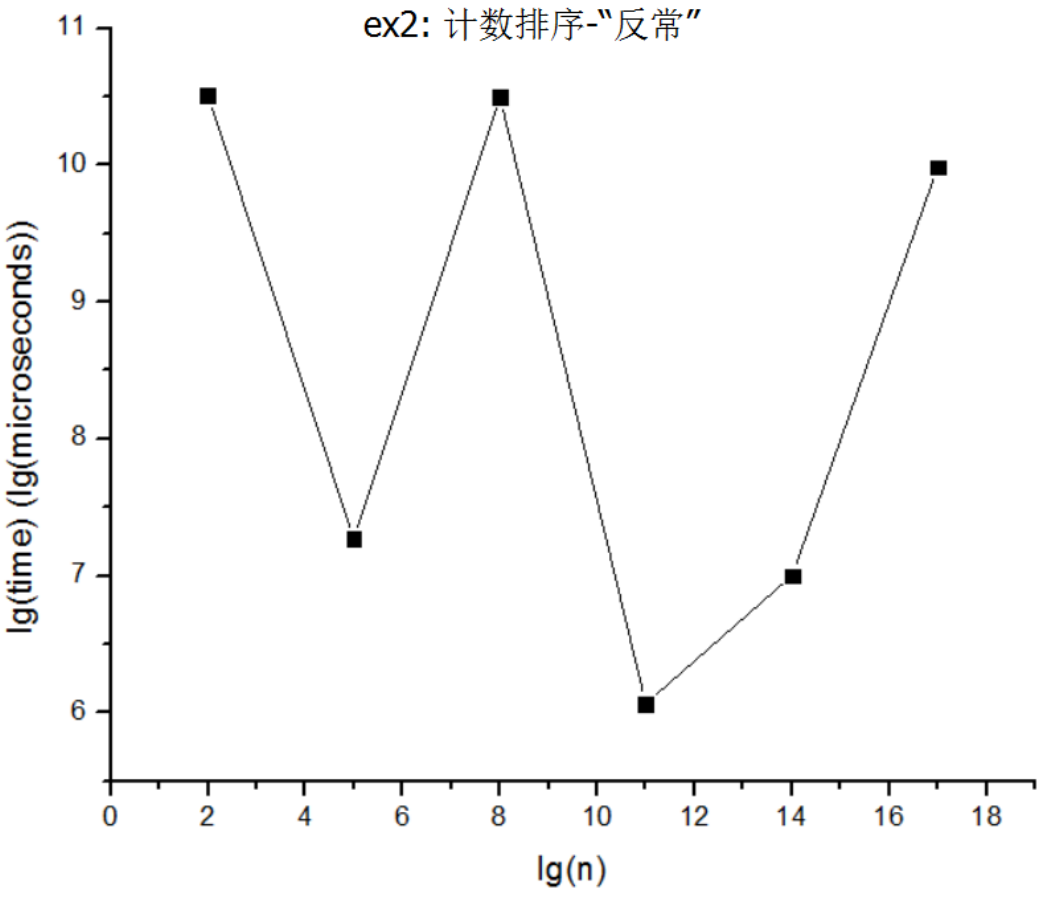
结果较为符合 $\Theta(n^2)$ 的期望。

iii. 计数排序：

计数排序排序的结果较为“反常”，尤其是当 n 比较小的时候：

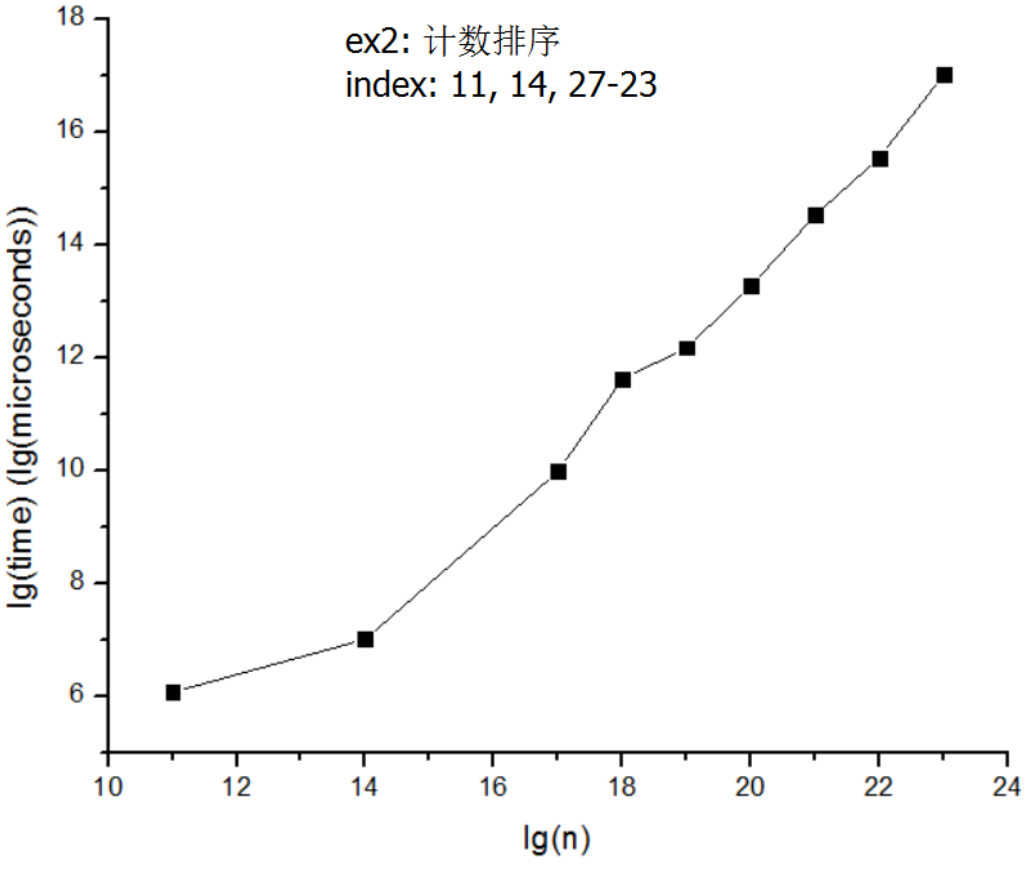
ex2:计数排序			
n	time(microseconds)	lg(n)	lg(time)
4	1,459	2	10.5108
32	154	5	7.26679
256	1,444	8	10.4959
2,048	67	11	6.06609
16,384	128	14	7
131,072	1,012	17	9.98299

于是为了减少小数据时，硬件策略对实验的影响，增加了 index=18,19,20,21,22,23 的排序数据（当然特地为此增加了随机数的生成）。得到以下数据：



看起来跟 $\Theta(k+n)$ 的期望有点矛盾。

ex2:计数排序			
n	time(microseconds)	lg(n)	lg(time)
4	1,459	2	10.5108
32	154	5	7.26679
256	1,444	8	10.4959
2,048	67	11	6.06609
16,384	128	14	7
131,072	1,012	17	9.98299
262,144	3,133	18	11.6133
524,288	4,611	19	12.1709
1,048,576	9,826	20	13.2624
2,097,152	23,491	21	14.5198
4,194,304	47,489	22	15.5353
8,388,608	131,973	23	17.0099



结果较为符合 $\Theta(k+n)$ 的期望。

为了分析数据少的时候产生“不合理”波动的原因，我为 counting_sort 函数中的过程分别测量了时间：

```
    public static void counting_sort(int[] A){
        long s;
        long e;
        s = System.nanoTime();

        int tmp;
        int l = A.length;
        B = new int[l];
        int[] C = new int[65535];
        e = System.nanoTime();
        System.out.println("init:81 " + (e-s)/1000 + "\tmicroseconds.");
        // t = (e-s)/1000;

        s = System.nanoTime();
        for(int i = 0; i < l; i++){
            C[A[i]-1]++;
        }
        // C[i] 的值即为大小为i-1的值的个数
        e = System.nanoTime();
        System.out.println("init:90 " + (e-s)/1000 + "\tmicroseconds.");


        s = System.nanoTime();
        for(int i = 1; i < 65535; i++){
            C[i] += C[i-1];
        }
        // C[i] 的值即为大小为小于等于i-1的值的个数
        e = System.nanoTime();
        System.out.println("init:65535 " + (e-s)/1000 + "\tmicroseconds.");

        // t += (e-s)/1000;

        s = System.nanoTime();
        for(int i = l-1; i >= 0; i--){
            B[C[A[i]-1]-1] = A[i];
            C[A[i]-1] --;
        }
        e = System.nanoTime();
        System.out.println("[][][] " + (e-s)/1000 + "\tmicroseconds.");
    }
```

结果：

```
Reaper@KZ:/mnt/d/USTC/algorithm/PB15111604-project1/ex2/source$ make counting
javac -encoding utf-8 Counting.java;
java Counting;
init:81 33      microseconds.
init:90 2       microseconds.
init:65535 2772 microseconds.
[][][] 8        microseconds.
index: 2
time: 6716      microseconds.

init:81 20      microseconds.
init:90 3       microseconds.
init:65535 125  microseconds.
[][][] 3        microseconds.
index: 5
time: 1535      microseconds.

init:81 28      microseconds.
init:90 15      microseconds.
init:65535 1446 microseconds.
[][][] 21       microseconds.
index: 8
time: 2720      microseconds.

init:81 31      microseconds.
init:90 148     microseconds.
init:65535 1377 microseconds.
[][][] 337      microseconds.
index: 11
time: 3662      microseconds.

init:81 15      microseconds.
init:90 267     microseconds.
init:65535 135  microseconds.
[][][] 188      microseconds.
index: 14
time: 2577      microseconds.
```

```
init:81 32      microseconds.
init:90 533     microseconds.
init:65535 81   microseconds.
[][][] 1011     microseconds.
index: 17
time: 4107      microseconds.

init:81 51      microseconds.
init:90 408     microseconds.
init:65535 70   microseconds.
[][][] 1651     microseconds.
index: 18
time: 3603      microseconds.

init:81 236     microseconds.
init:90 715     microseconds.
init:65535 71   microseconds.
[][][] 3781     microseconds.
index: 19
time: 6474      microseconds.

init:81 706     microseconds.
init:90 1364    microseconds.
init:65535 64   microseconds.
[][][] 8821     microseconds.
index: 20
time: 12364     microseconds.

init:81 1149    microseconds.
init:90 4018    microseconds.
init:65535 51   microseconds.
[][][] 20296    microseconds.
index: 21
time: 28340     microseconds.

init:81 2913    microseconds.
init:90 7592    microseconds.
init:65535 94   microseconds.
[][][] 63999    microseconds.
index: 22
time: 76572     microseconds.

init:81 7686    microseconds.
init:90 13789   microseconds.
init:65535 120  microseconds.
[][][] 114298   microseconds.
index: 23
```

发现主要的问题在于 init:65535 部分，即：

```
92          s = System.nanoTime();
93          for(int i = 1; i < 65535; i++){
94              C[i] += C[i-1];
95          }
96          // C[i] 的值即为大小为小于等于i-1的值的个数
97          e = System.nanoTime();
98          System.out.println("init:65535 " + (e-s)/1000 + "\tmicroseconds.");
```

这一部分，数据少的时候这里波动大，而当 index>=17,即 n>=131072 时则为波动极小的常数。

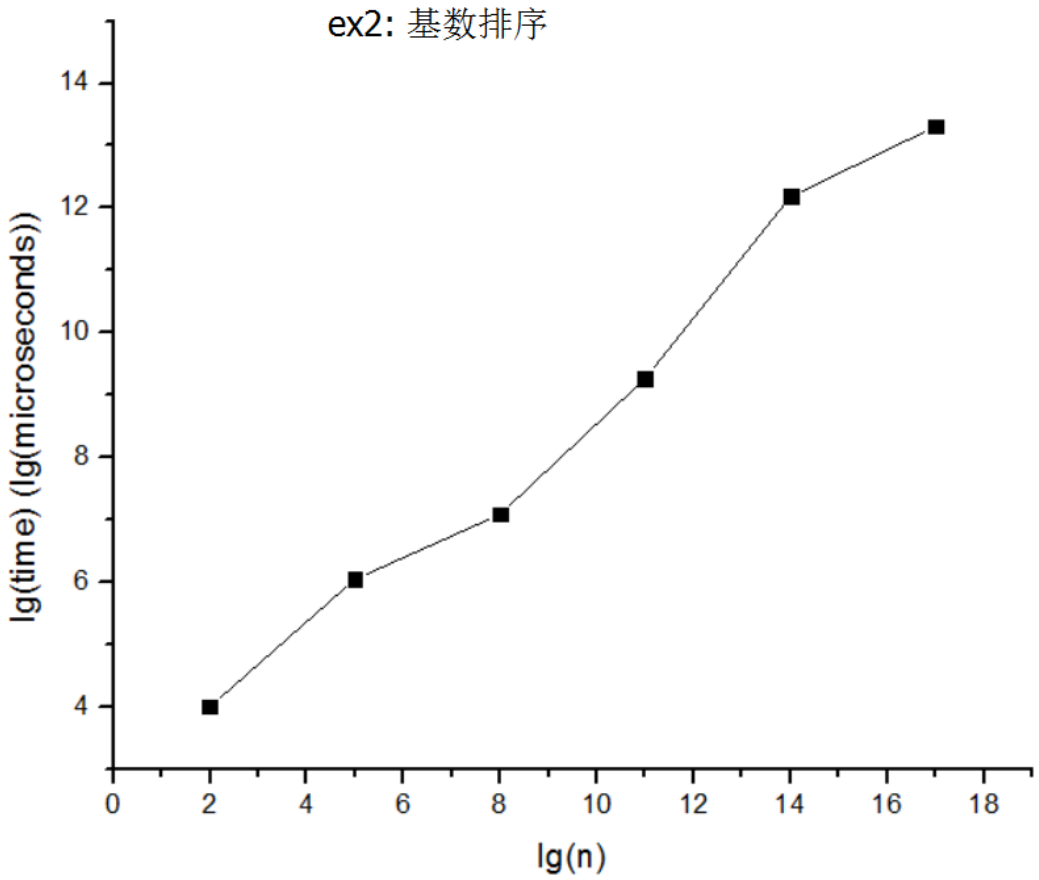
据我的分析，可能是因为：

1. 当 n 小的时候，65535 次 C[i]存取和“+=”的操作的影响远大于 n 次其他操作对总体时间的影响。
2. 至于波动，我分析可能是因为硬件客观存在的 cache 环境以及 cache 策略的影响。正是硬件这样客观存在的 cache 环境以及 cache 策略的原因，导致了前期大量波动的 cache miss，也就导致了时间的波动。

基数排序：

ex2:基数排序			
n	time(microseconds)	lg(n)	lg(time)
4	16	2	4
32	63	5	5.97728
256	137	8	7.09803
2,048	707	11	9.46557
16,384	5,352	14	12.3859
131,072	10,636	17	13.3767

得图：

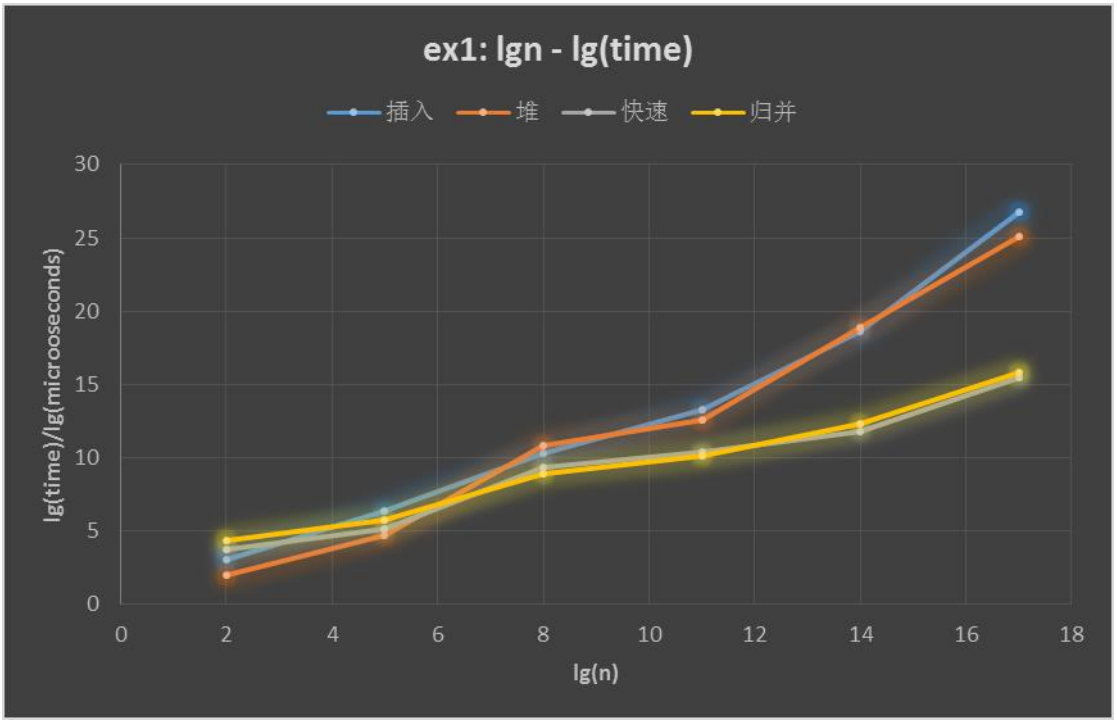


较为符合 $\Theta(d(n+k))$ 的期望。

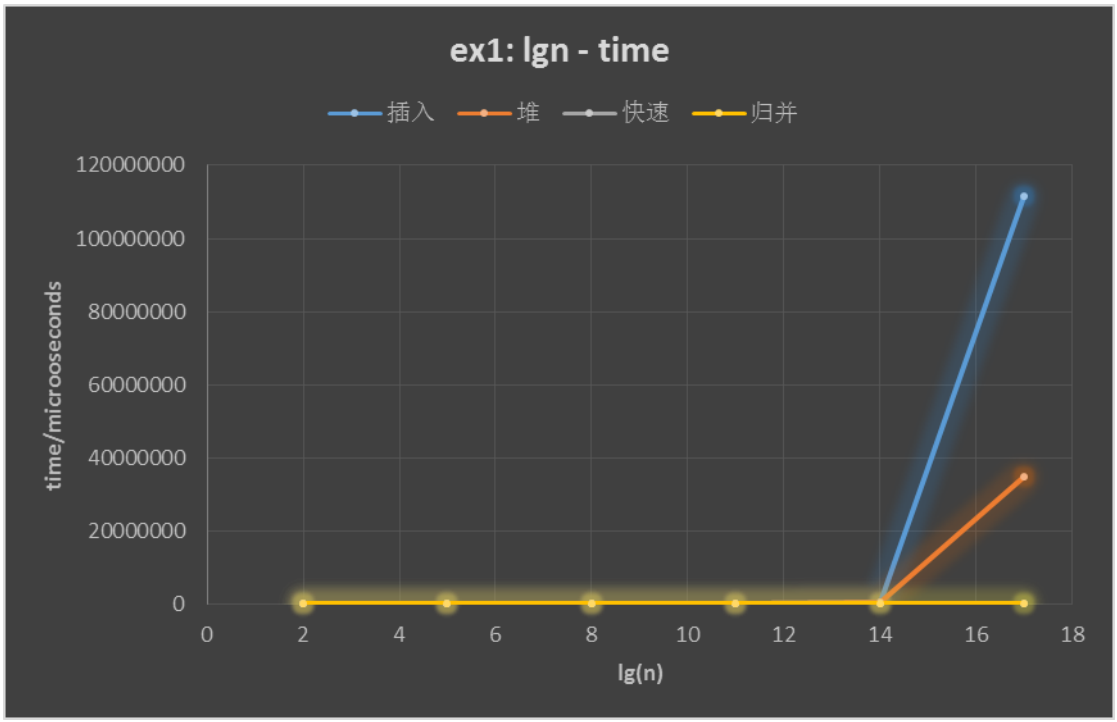
至于为什么用了计数排序却没有像刚才那么反常，应该是因为对应的计数排序的 k 是 16，而上一次是 65535，所以这里 k 本身没有比 n 大很多的情况。所以比较正常。

ex1 综合分析：

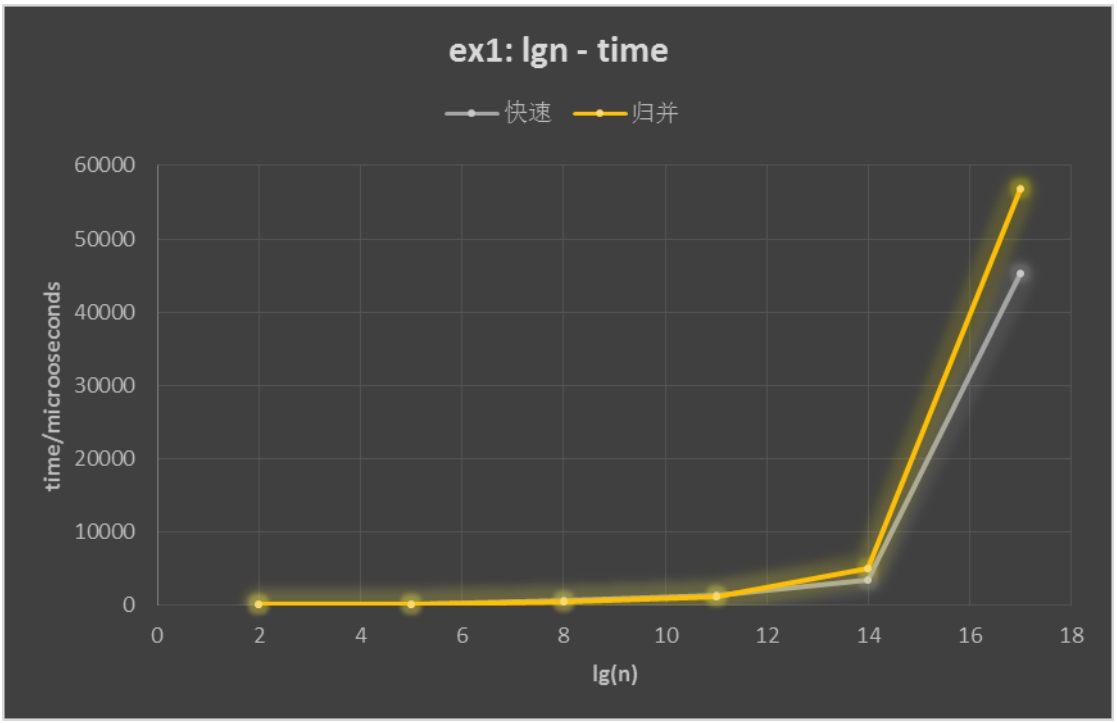
lg(n) – lg(time):



lg(n) – time:



lg(n) – time——归并与快速：



从以上三图可预测，

在 n=4 时，消耗时间由短到长为：

堆　　插入　　快速　　归并

在 n=100 附近，四个算法时间相近，之后的速度分为两个梯队：

快的是　快速，归并

慢的是　堆，插入

之后随着 n 的增大，每个算法之间差距越来越大，

最终到了 n=10000 附近，差距愈发明显消耗时间由短到长为：

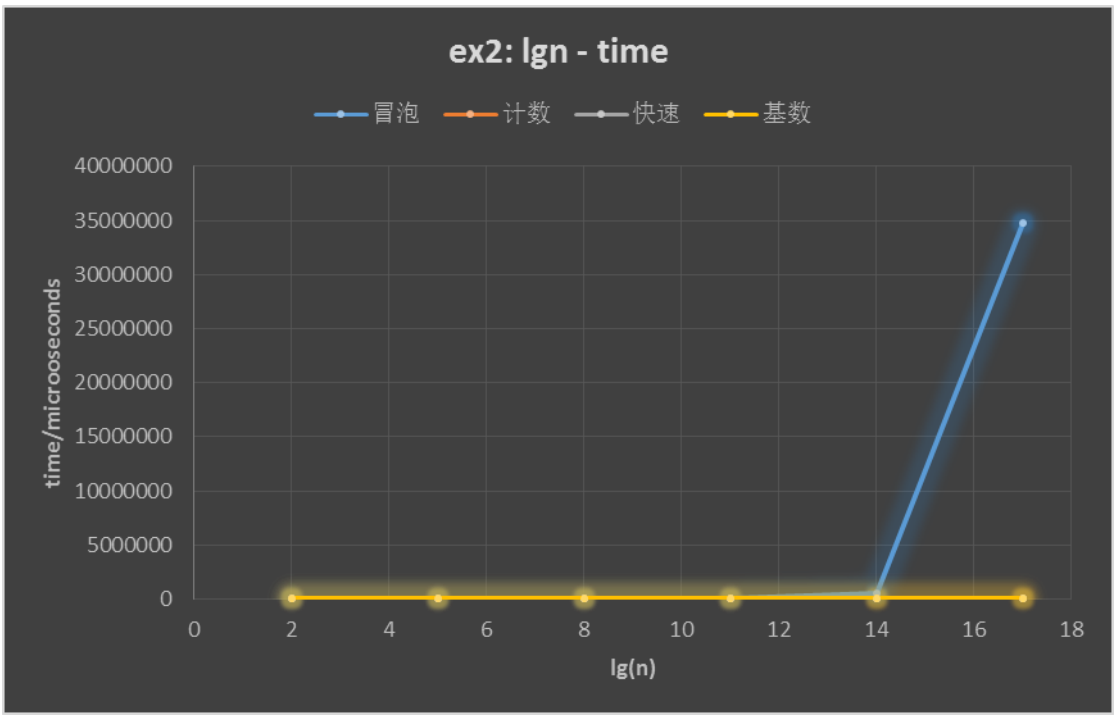
快速　　归并　　堆　　插入

综上，当数据少时，性能差的不多，可以选择堆排序。

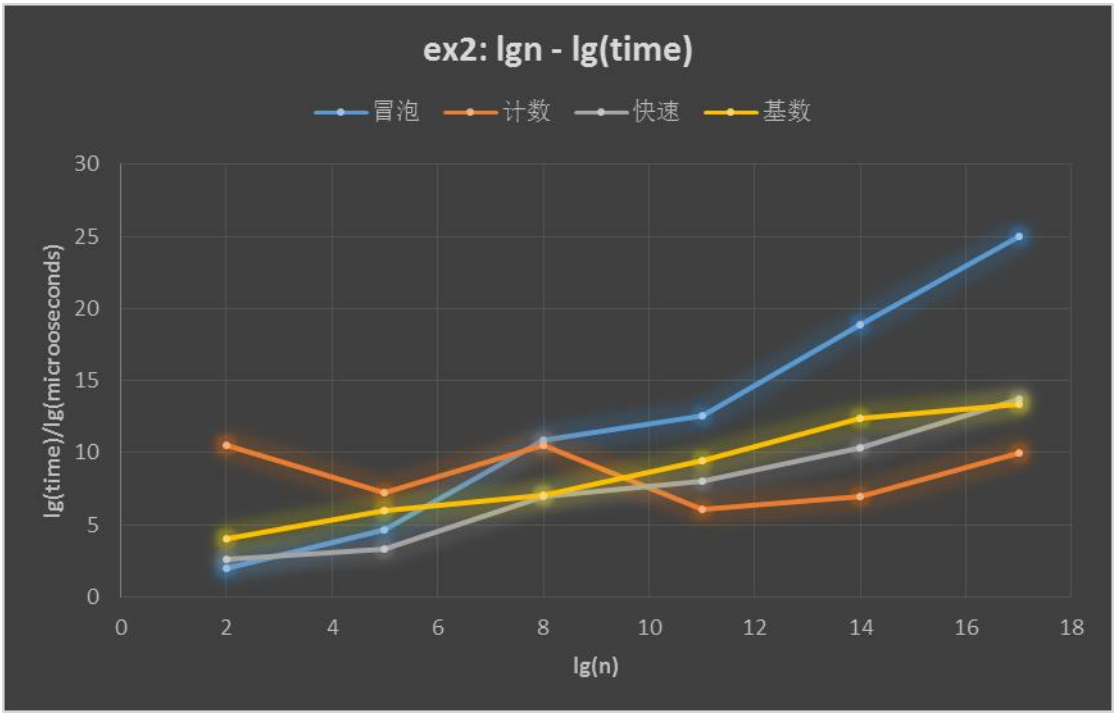
而当数据较多时，除了极端情况下，都应该选择快速排序。

ex2 综合分析：

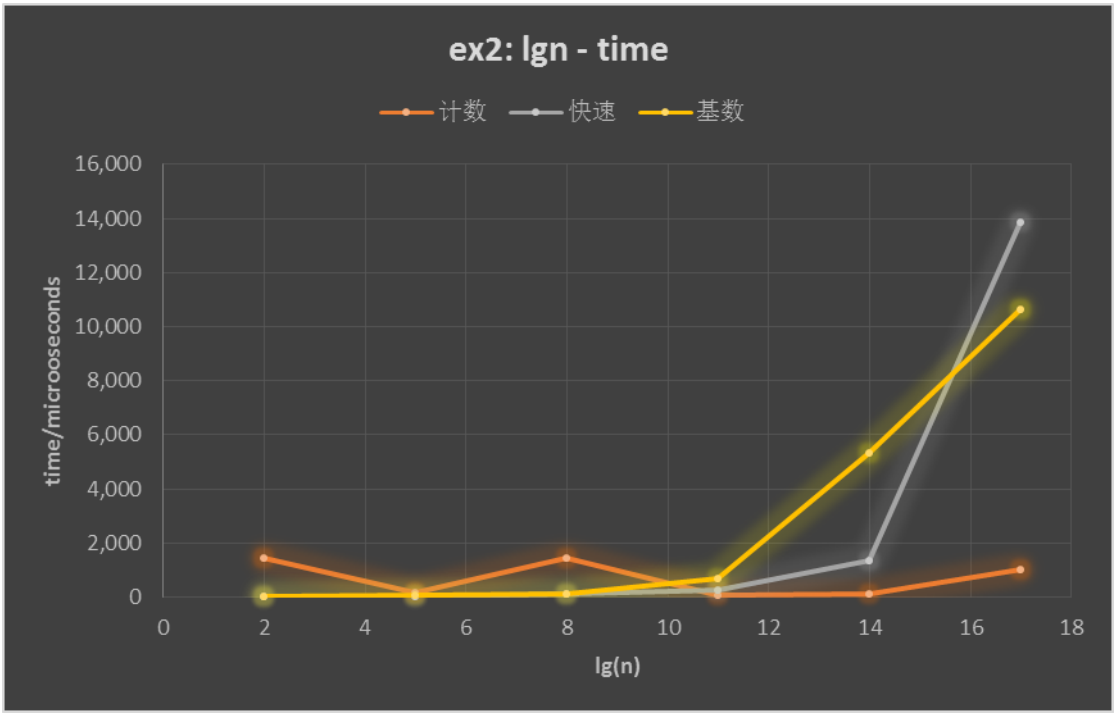
lgn – time:



lgn-lg(time):



lgn-time:



以上三图可预测，

在 n 比较小时，计数排序会有较大的时间波动。

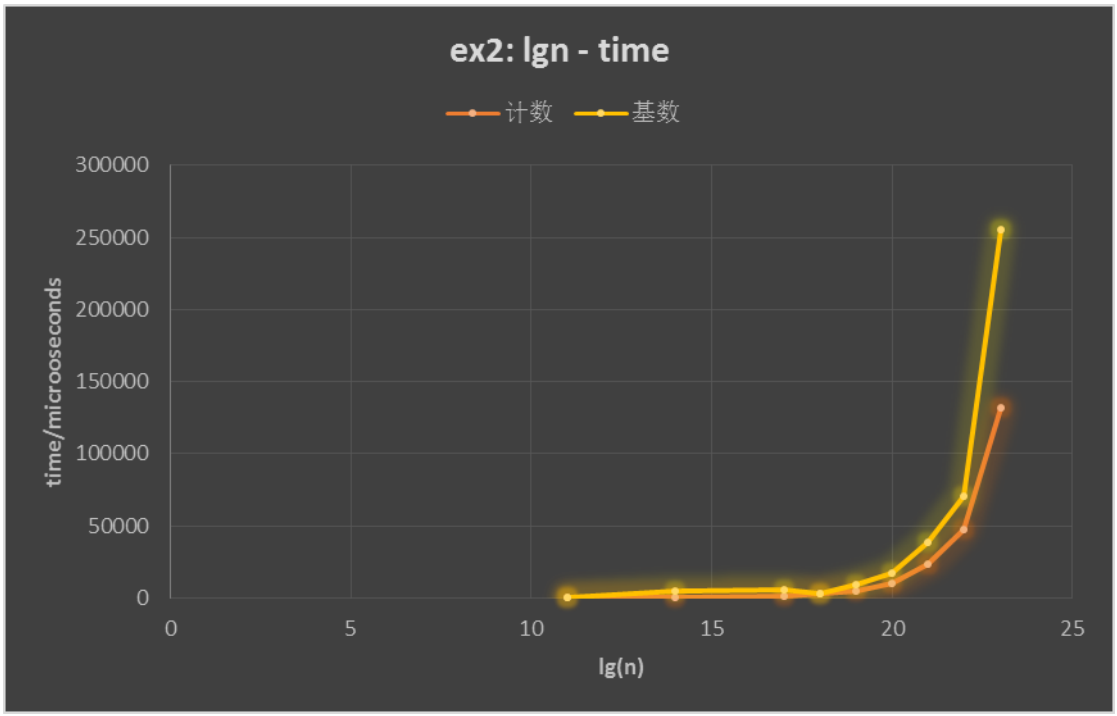
在 n=4 时，消耗时间由短到长为：

冒泡　　快速　　基数　　计数

在 n 较大时，冒泡排序完全无法与其他三个比较。

而对于快速，基数，计数三个算法，随着 n 的增大，刚开始是快速排序占优势，后来随着 n 的增大，快速排序由于毕竟是 nlgn 的量级，所以被拉开距离。而基数排序与计数排序之间，计数排序在没有了 n 较小时的波动影响之后，占据优势，比基数排序还要快很多。

考虑到更大的 n 时的情况，我特地将基数排序的数据扩大到了 2^23，与计数一决雌雄。得到下图。说明后面计数排序确实占优势。



所以在数据的取值条件适合的形况下：
当数据很少（如不到 20）时，为了方便可以选择冒泡，
再大一点应该选择快速排序，
到了 n=1000 附近开始，就可以选择计数排序了。

7. 实验心得

a) 首先，通过本次实验，加深了对各个排序算法的理解，这是最重要的。尤其是通过实现各个算法，发现了很多以前没有考虑到的细节。这是最大的收获。
b) 其次，通过本次实验，对排序算法具体条件下的性能选择有了更好的把握，之前只知道具体的渐进性能函数，而不知道具体的性能。
c) 再其次，通过本次实验，我对刚学的 java 有了进一步的理解与掌握。在写本次实验之前我都没有超过 20 行的 java 代码。
d) 再再其次，通过本次实验，我学会了合理的、优美的绘图技巧。实验做到了一半才在谷歌上了解到 excel 绘图能力的强大，（所以前半部分的图都是用之前大物实验用的 origin 画的，后面使用 excel 画的，其优美之差异真是天壤之别。）
e) 最后，感谢可爱哒助教读到这里，感谢的同时心疼一下。。。

声明：由于过程中生成的 2^23 大小的数据导致压缩包过大，不方便 email，所以只保留了前面的 2^17 部分，助教可以在 ex1/source/下 make generage 生成。