1 运行时间测量——比较三种排序算法

本实验假设所有代码文件放在同一文件夹中。请按照要求，产生可执行文件，并给出每一个步骤的截图。以下要求使用 Linux 或者 Windows 的 GCC 工具。参考https://www.overleaf.com/read/dvrnfhdtgcnb 。

1.1 计数排序法

以下包括代码和操作。

1.1.1 counting_sort.h 文件

```
1 void counting_sort(int *, int, int *);
```

1.1.2 counting_sort.c 文件

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "counting_sort.h"
4
5  const int MAX_ELEMENT = RAND_MAX;
6  typedef unsigned long long uLL;
7
8  void counting_sort(int *a, int n, int *b)
9  /*给定数组首元素地址a，数组长度n，对数组进行计数排序，
10 并把排序结果存在以b为首元素地址的数组中，
11 用户须确保以b开头的存储空间足够存储n个int型元素，
12 只适用于对一定范围内的非负整数进行排序*/
13 {
14    // 分配数组，用于存储以a为首元素地址的数组中各个元素出现的次数
15    // 存储的数据为int型，数据的个数为LL型
16    uLL *occur_times = (uLL *)malloc(sizeof(uLL) * (MAX_ELEMENT
          + 1));
17    if(!occur_times)
18    {
19      printf("insufficient memory\n");
20      exit(EXIT_FAILURE);
```

```
21      }
22      // 各个元素出现的次数初始化为零
23      for(int element_id = 0; element_id <= MAX_ELEMENT + 1;
            element_id++)
24      {
25         occur_times[element_id] = 0; //
26      }
27      // 统计各个元素出现的次数（计数）
28      for(uLL element_location_index = 0; element_location_index
            < n; element_location_index++)
29      {
30         occur_times[a[element_location_index]]++; //
31      }
32      // 分区
33      // 任意给定元素e，得到小于e的元素的个数
34      // 位置为LL型
35      uLL *occur_positions = (uLL *)malloc(sizeof(uLL) *
            (MAX_ELEMENT + 1));
36      if(!occur_positions)
37      {
38        printf("insufficient memory\n");
39        exit(EXIT_FAILURE);
40      }
41      // occur_position[i]也理解为此刻i应该填入的位置
42      occur_positions[0] = 0; // 小于0的元素个数
43      for(int element_location_index = 1; element_location_index
            <= MAX_ELEMENT + 1; element_location_index++)
44      {
45         occur_positions[element_location_index] =
               occur_positions[element_location_index-1] +
               occur_times[element_location_index-1]; //
               递推，比i小的最大元素最早出现位置+它在输入数组中出现的次数
46      }
47      // 填入
48      for(uLL element_location_index = 0; element_location_index
            < n; element_location_index++) // 位置为LL型
49      {
50         int v = a[element_location_index]; // 待填入的值
51         b[occur_positions[v]++] = v; //
```

```
            把v填入之后，待填入的目标地址+1
52      }
53      free(occur_times);
54      free(occur_positions);
55  }
```

1.1.3 main1.c 文件

```
1   #include <stdio.h>
2   #include <limits.h>
3   #include <time.h>
4   #include <stdlib.h>
5
6   #include "counting_sort.h"
7
8   #define DEBUG_MODE
9
10  typedef unsigned long long uLL;
11
12  int main()
13  {
14      printf("ULLONG_MAX: %llu\n", ULLONG_MAX);
15  #ifdef DEBUG_MODE
16      printf("RAND_MAX: %d\n", RAND_MAX);
17  #endif
18      int right_shift = 33;
19      uLL arr_len = ULLONG_MAX >> right_shift;
20      printf("The length of a random sequence to be applied
            counting sort on is: %llu\n", arr_len);
21      // 待排序的数组
22      int *arr = (int*)malloc(sizeof(int) * arr_len);
23      if(!arr)
24      {
25          printf("insufficient memory for %llu ints\n", arr_len);
26          exit(EXIT_FAILURE);
27      }
28      srand(time(NULL));
29      for(uLL i = 0; i < arr_len; i++)
```

```
30      {
31          arr[i] = rand();
32      }
33  #ifdef DEBUG_MODE
34      printf("the random sequence to be sorted:\n");
35      for(uLL i = 0; i < arr_len; i++)
36      {
37          printf("%d\t", arr[i]);
38      }
39      printf("\n");
40  #endif
41      // 用于存放排序结果
42      int *sorted_array = (int *)malloc(sizeof(int) * arr_len);
43      if(!sorted_array)
44      {
45          printf("insufficient emory for %llu ints\n", arr_len);
46          exit(EXIT_FAILURE);
47      }
48
49      // 待排序数组的元素个数
50      int num_of_elements = arr_len;
51
52      // 计数排序
53      counting_sort(arr, num_of_elements, sorted_array);
54
55  #ifdef DEBUG_MODE
56      // 输出
57      printf("the result in ascending order:\n");
58      for(uLL i = 0; i < num_of_elements; i++)
59      {
60          printf("%d\t", sorted_array[i]);
61      }
62      printf("\n");
63  #endif
64      // 回收内存
65      free(arr);
66      free(sorted_array);
67      return 0;
68  }
```

1.1.4 步骤

1. 在 main1.c 中添加代码，用于测试它进行排序所花的时间。注意不能把输入输出语句运行所花的时间包括在内，被添加的代码不能落在相邻的 #ifdef DEBUG_MODE 和 #endif 之间，并且在程序运行的最后阶段输出这个时间长度。（注意文档开头所给的网址。）

2. 把 right_shift 的声明行中的初始值修改为 58，编译运行产生可执行文件，并把生成的可执行文件命名为 main1_1.exe。运行 main1_1.exe 并记下输出结果。

3. 把上一步提及的初始值依次修改为 59、60、61，分别重做上述实验，并记下每一次的输出结果。

4. 把 main1.c 中 #define DEBUG_MODE 所在的行注释掉。

5. 接上一步，把 right_shift 的声明行中的初始值修改为 33，编译运行产生可执行文件，并且把可执行文件命名为 main1_2.exe。运行 main1_2.exe 并记下输出结果（内存不足的情况忽略不计）。

6. 把上一步提及的初始值依次修改为 34, 35……40, 重做上述实验, 并记下每一次的输出结果（内存不足的情况忽略不计）。

7. (a) 针对上一步 main1_2.exe 的输出结果，绘制表格。表格第一行列出先后被排序的随机序列长度的值,第二行列出 main1_2.exe 的排序过程所花的时间。

   (b) 请根据上述表格画出散点图并给出大致的拟合曲线（可以纯手工进行），然后简要说明数据的趋势。

1.2 冒泡排序法

以下包括代码和操作。

1.2.1 bubble_sort.h 文件

```
1 void bubble_sort(int *, int);
```

1.2.2 bubble_sort.c 文件

```c
#include "bubble_sort.h"

/*冒泡排序*/
void bubble_sort(int *a, int n)
{  //逐步求更长的升序序列
   for(int i = 0; i < n; i++) // i位置之前元素已就位
   {
      for(int j = n - 1; j > i; j--) // j位置元素考虑前移
      {
         if(a[j - 1] > a[j]) // 违反升序要求
         {
            // 交换
            int temp = a[j - 1];
            a[j - 1] = a[j];
            a[j] = temp;
         }
      }
   }
}
```

1.2.3 main2.c 文件

```c
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <stdlib.h>

#include "bubble_sort.h"

#define DEBUG_MODE

typedef unsigned long long uLL;

int main()
{
```

```
14     printf("ULLONG_MAX: %llu\n", ULLONG_MAX);
15 #ifdef DEBUG_MODE
16     printf("RAND_MAX: %d\n", RAND_MAX);
17 #endif
18     int right_shift = 47;
19     uLL arr_len = ULLONG_MAX >> right_shift;
20     printf("The length of a sequence to be applied counting
           sort on is: %llu\n", arr_len);
21     // 待排序的数组
22     int *arr = (int*)malloc(sizeof(int) * arr_len);
23     if(!arr)
24     {
25         printf("insufficient emory for %llu ints\n", arr_len);
26         exit(EXIT_FAILURE);
27     }
28     srand(time(NULL));
29     for(uLL i = 0; i < arr_len; i++)
30     {
31         arr[i] = rand();
32     }
33 #ifdef DEBUG_MODE
34     printf("the random sequence to be sorted:\n");
35     for(uLL i = 0; i < arr_len; i++)
36     {
37         printf("%d\t", arr[i]);
38     }
39     printf("\n");
40 #endif
41     // 用于存放排序结果
42     int *sorted_array = (int *)malloc(sizeof(int) * arr_len);
43     if(!sorted_array)
44     {
45         printf("insufficient emory for %llu ints\n", arr_len);
46         exit(EXIT_FAILURE);
47     }
48
49
50     for(uLL i = 0; i < arr_len; i++)
51     {
```

```
52          sorted_array[i] = arr[i];
53      }
54
55      // 待排序数组的元素个数
56      int num_of_elements = arr_len;
57
58      // 冒泡排序
59      bubble_sort(sorted_array, num_of_elements);
60
61  #ifdef DEBUG_MODE
62      // 输出
63      printf("the result in ascending order:\n");
64      for(uLL i = 0; i < num_of_elements; i++)
65      {
66          printf("%d\t", sorted_array[i]);
67      }
68      printf("\n");
69  #endif
70      // 回收内存
71      free(arr);
72      free(sorted_array);
73  }
```

1.2.4 步骤

1. 在 `main2.c` 中添加代码，用于测试它进行排序所花的时间。注意必须把序列复制的时间包括在内，不能把输入输出语句运行所花的时间包括在内，被添加的代码不能落在相邻的 `#ifdef DEBUG_MODE` 和 `#endif` 之间，并且在程序运行的最后阶段输出这个时间长度。（注意文档开头所给的网址。）

2. 把 `right_shift` 声明行中的初始值修改为 `58`，编译运行产生可执行文件，并且把生成的可执行文件命名为 `main2_1.exe`。运行 `main2_1.exe` 并记下输出结果。

3. 把上一步提及的初始值依次修改为 `59`、`60`、`61`，分别重做上述实验，并记下每一次的输出结果。

4. 把 `main2.c` 中 `#define DEBUG_MODE` 所在的行注释掉。

5. 接上一步，把 `right_shift` 声明行中的初始值修改为 `47`，编译运行产生可执行文件，并且把可执行文件命名为 `main2_2.exe`。运行 `main2_2.exe` 并记下输出结果。

6. 把上一步提及的初始值依次修改为 `48`, `49`……`55`, 重做上述实验，并记下每一次的输出结果。

7. (a) 针对上一步 `main2_2.exe` 的输出结果，绘制表格。表格第一行列出先后被排序的随机序列的长度，第二行列出 `main2_2.exe` 的排序过程所花的时间。

   (b) 请根据上述表格画出散点图并给出大致的拟合曲线（可以纯手工进行），然后简要说明数据的趋势。

1.3 快速排序

以下包括代码和操作。

1.3.1 `quick_sort.h` 文件

```
// 对迭代器的子表arr[low ... high]进行排序
void quick_sort(int *, int, int);
/*
调用时的前置值: low = 1 (原序列第一个元素的下标), high =
    (原序列最后一个元素的下标)
对子表arr[low, high]做快速排序
*/
```

1.3.2 `quick_sort.c` 文件

```
#include "quick_sort.h"

#include <stdlib.h>

// 把子表arr[low ... high]一分为二，并返回轴枢的位置
int partition(int *arr, int low, int high)
```

```
7  // 在排序中arr[0]被用于临时存放轴枢
8  {
9     //
          注意到快速排序中，轴枢的选择是非常有技巧的，这里只为了说明问题，因而简单处理
10    int rand_loc = low + rand() % (high - low + 1); //
          在low和high（含端点）之间随机选择一个位置
11
12    //
          交换是为了确保轴枢的选择具有随机性，从而加强了排序效率的稳定性，避免最坏情况造成过大的效率问题
13    int temp = arr[rand_loc];
14    arr[rand_loc] = arr[low];
15    arr[low] = temp;
16    // 交换后的数组内容不变，a[low]可以是之前数组的任何一个元素
17
18    int pivot_key = arr[0] = arr[low]; //
          arr[low]的值作为轴枢值存在了arr[0]，因此arr[low]这个空位实际上成为了闲置空间
19    //
          我们将把这个闲置空间作为轴枢，把一切>=轴枢的放右边，把一切<=轴枢的放左边，
20    // 最后把pivot_key往这个闲置空间一放就可以了
21    // 比如数组arr[]={*, 3, 3, 1, 2, 5,
          3}，如果我们选择了a[1]作为轴枢，
22    // 就会得到arr[]={3, *, 3, 1, 2, 5,
          3}，注意到此时a[1]的值已经保存在a[0]，并且a[1]成为闲置空间
23    // 那么，a[2]和a[6]存储的3可以随意放在轴枢(*)的任何一侧
24
25
26    //
          注意在分区过程中，我们一直都有一个闲置的位置，直到最后填入pivot_key为止
27    // 以下运行中，要维护：
28    // (1) low指向闲置空间，
29    // (2) low左侧的元素是顶多不超过pivot_key，
30    // (3) high右侧的元素最低不低于pivot_key
31    while(low < high) // 仍有元素未确定分区
32    {
33       // 逐个检查，尽力确定高半区的元素
34       while(low < high && arr[high] >= pivot_key) //
             关键字等于pivot_key时，元素可以放在任何一侧，所以这里可以取=号，
35                                                   //
                                                        而且也必须取等号，否则high指针可能无法移动
```

```
36
37          {
38              --high; //
                    跳过（注意这里的编程技巧，物理上移动指针，从而做到逻辑上移动元素）
39          }
40          //
                此时arr[high]必小于pivot_key，因此，它的值一定要放在轴枢(*)的左边
41          //
                如何把a[high]的元素移动到轴枢(*)的左边？注意到轴枢(*)实际上是闲置空间
42          arr[low] = arr[high]; // 对闲置空间进行写入操作不会丢失有用数据
43                                // 并且写入操作执行后，a[high]成为了闲置空间
44
45          // 逐个检查，尽力确定低半区的元素
46          while(low < high && arr[low] <= pivot_key) //
                关键字等于pivot_key时，元素可以放在任何一侧，所以这里可以取=号，
47                                                    //
                                                        而且也必须取等号，否则low指针可能无法移动
48          {
49              ++low; //
                    跳过（注意这里的编程技巧，物理上移动指针，从而做到逻辑上移动元素）
50          }
51          //
                此时arr[low]必大于pivot_key，因此，它的值一定要放在轴枢(*)的右边
52          //
                如何把a[low]的元素移动到轴枢(*)的右边？注意到在上一步，a[high]成为了闲置空间
53          arr[high] = arr[low]; // 对闲置空间进行写入操作不会丢失有用数据
54                                // arr[low]又成为了闲置空间
55      }
56      // 此时low==high，左右两侧的元素已经安放好
57      arr[low] = arr[0]; // 把轴枢的值放进当前的闲置空间arr[low]
58                         // arr[0]又成为了闲置空间
59      return low; // 返回轴枢的位置
60  }
61
62  // 对迭代器的子表arr[low ... high]进行排序
63  void quick_sort(int *arr, int low, int high)
64  /*
65  调用时的前置值：low = 1（原序列第一个元素的下标），high =
        （原序列最后一个元素的下标）
```

```
对子表arr[low, high]做快速排序
*/
{
    if(low < high)
    {
        int pivot_loc = partition(arr, low, high);
        quick_sort(arr, low, pivot_loc - 1);
        quick_sort(arr, pivot_loc + 1, high);
    }
}
```

1.3.3 main3.c 文件

```
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <stdlib.h>

#include "quick_sort.h"

#define DEBUG_MODE

typedef unsigned long long uLL;

int main()
{
    printf("ULLONG_MAX: %llu\n", ULLONG_MAX);
#ifdef DEBUG_MODE
    printf("RAND_MAX: %d\n", RAND_MAX);
#endif
    int right_shift = 58;
    uLL arr_len = ULLONG_MAX >> right_shift;
    printf("The length of a sequence to be applied counting
        sort on is: %llu\n", arr_len);
    // 待排序的数组
    int *arr = (int*)malloc(sizeof(int) * arr_len);
    if(!arr)
    {
```

```
25          printf("insufficient emory for %llu ints\n", arr_len);
26          exit(EXIT_FAILURE);
27      }
28      srand(time(NULL));
29      for(uLL i = 0; i < arr_len; i++)
30      {
31          arr[i] = rand();
32      }
33  #ifdef DEBUG_MODE
34      printf("the random sequence to be sorted:\n");
35      for(uLL i = 0; i < arr_len; i++)
36      {
37          printf("%d\t", arr[i]);
38      }
39      printf("\n");
40  #endif
41      // 用于存放排序结果
42      int *sorted_array = (int *)malloc(sizeof(int) * (arr_len +
            1));
43      if(!sorted_array)
44      {
45          printf("insufficient emory for %llu ints\n", arr_len +
                1);
46          exit(EXIT_FAILURE);
47      }
48
49
50      for(uLL i = 1; i < arr_len + 1; i++)
51      {
52          sorted_array[i] = arr[i - 1];
53      }
54
55      // 待排序数组的元素个数
56      int num_of_elements = arr_len;
57
58      // 快速排序
59      quick_sort(sorted_array, 1, num_of_elements);
60
61  #ifdef DEBUG_MODE
```

```
62      // 输出
63      printf("the result in ascending order:\n");
64      for(uLL i = 0; i < num_of_elements; i++)
65      {
66          printf("%d\t", sorted_array[i + 1]);
67      }
68      printf("\n");
69  #endif
70      // 回收内存
71      free(arr);
72      free(sorted_array);
73  }
```

1.3.4 步骤

1. 在 main3.c 中添加代码，用于测试它进行排序所花的时间。注意必须把序列复制的时间包括在内，不能把输入输出语句运行所花的时间包括在内，被添加的代码不能落在相邻的 #ifdef DEBUG_MODE 和 #endif 之间，并且在程序运行的最后阶段输出这个时间长度。（注意文档开头所给的网址。）

2. 把 right_shift 声明行中的初始值修改为 58，编译运行产生可执行文件，并且把产生的可执行文件命名为 main3_1.exe。运行 main3_1.exe 并记下输出结果。

3. 把上一步提及的初始值依次修改为 59、60、61，分别重做上述实验，并记下每一次的输出结果。

4. 把 main3.c 中 #define DEBUG_MODE 所在的行注释掉。

5. 接上一步，把 right_shift 声明行中的初始值修改为 37，编译运行产生可执行文件，并且把可执行文件命名为 main3_2.exe。运行 main3_2.exe 并记下输出结果。

6. 把上一步提及的初始值依次修改为 38, 39……45，重做上述实验，并记下每一次的输出结果。

7. 统计分析。

(a) 针对上一步 `main3_2.exe` 的输出结果，绘制表格。表格第一行列出先后被排序的随机序列的长度，第二行列出 `main3_2.exe` 的排序过程所花的时间。

(b) 请根据上述表格画出散点图并给出大致的拟合曲线（可以纯手工进行），然后简要说明数据的趋势。

1.4 实验对比

对比第1.1节中的步骤7、第1.2节中的步骤7和第1.3节中的步骤7中的曲线和表格，我们可以看出什么规律?

1.5 实验报告写作要求

1. 步骤详细；
2. 表述简明；
3. 图文并茂；
4. 逻辑流畅。