Project 1 report

实验内容

- 排序n个元素，元素为随机生成的0到$2^{15}-1$之间的整数，n的取值为： 2^3, 2^6, 2^9, 2^{12}, 2^{15}, 2^{18}。
- 实现以下算法：堆排序，快速排序，归并排序，计数排序。

实验配置与环境

- 操作系统：Windows 11 专业版（version 21H2）
- 处理器：11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- 时钟主频：2.80 GHz
- 编译器：gcc version 8.1.0

实验步骤

1. **编写排序程序**
    - **堆排序**：采用非递归的MAX_HEAPIFY，主要思想为，**调整堆时，只有不合法的节点和与之交换的下层节点可能不合法**，因此具体实现如下：（各步骤已注释）

```
void max_heapify(int arr[], int idx, int size){
    int lson = idx * 2 + 1;
    int rson = lson + 1;
    while(lson < size){
        /* find the largest one between parent, lson and ron */
        int largest = lson;
        if(rson < size && arr[lson] < arr[rson]){
            largest = rson;
        }
        if(arr[idx] > arr[largest]){
            largest = idx;
        }
        // if parent is the largest, heap is alreay a max_heap
        if(largest == idx) break;
        swap(arr, idx, largest);
        /* go down to adjust heap */
        idx = largest;
        lson = idx * 2 + 1;
        rson = lson + 1;
    }
}
```

      其余堆排序代码基本与书上代码无明显差别，具体请参考src中的heapsort.cpp

    - **归并排序**：将归并排序改为非递归，**只需每次对数组进行逐步变长的划分，在每个划分的内部merge即可**：

```
void merge_sort(int arr[], int p, int r){
    int left, mid, right;
    /* for each step, the length should double */
    for(int i = 1; i < r - p + 1; i *= 2){
        for(left = p ; left < r - p + 1 - i; left = right + 1){
            mid = left + i - 1; // get middle elem
            right = mid + i;
            if (right > r) right = r; // right may be larger than r
            merge(arr, left, mid, right);
        }
    }
}
```

merge函数的代码与书中无明显差别，具体请参考src中的mergesort.cpp

- **快速排序**：将快速排序改为非递归，需要使用到栈进行运算。**每次将划分后的两个数组的最左、最右位置压入栈中**：

```
void quick_sort(int arr[], int p, int r){
    rec.push(record(p, r));
    while(!rec.empty()){
        /* get a record to do partition */
        record tmp = rec.pop();
        int q = random_partition(arr, tmp.left, tmp.right);
        /* push the left and right into stack */
        if(tmp.left < q-1){
            rec.push(record(tmp.left, q-1));
        }
        if(tmp.right > q+1){
            rec.push(record(q+1, tmp.right));
        }
    }
}
```

其中，record类记录了最左、最右位置，如下：

```
class record{
    public:
    int left;
    int right;
    record(int l, int r){
        left = l;
        right = r;
    }
};
```

C++自带的stack库中的栈类运行效率太低，因为其冗余较多。为了提高效率，我自行实现了一个栈类，详细内容请参考我的代码。

在划分过程中，我使用随机划分来优化运行效率：

```
int random_partition(int arr[], int p, int r){
    /* get a location to be the key */
    int idx = rand() % (r - p + 1) + p;
```

```
    /* swap the last one and the key */
    swap(arr, r, idx);
    int key = arr[r];
    int i = p - 1;
    for(int j = p; j < r; j++){
        if(arr[j] <= key){
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i+1, r);
    return i + 1;
}
```

- **计数排序**：计数排序算法与教材中无明显差异，具体实现如下：

```
void counting_sort(int arr[], int sorted_arr[], int k, int n){
    int *count = new int[k];
    for(int i = 0; i < k; i++){
        count[i] = 0;
    }
    for(int i = 0; i < n; i++){
        count[arr[i]]++;
    }
    for(int i = 1; i < k; i++){
        count[i] += count[i-1];
    }
    for(int i = n - 1; i >= 0; i--){
        sorted_arr[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }
}
```

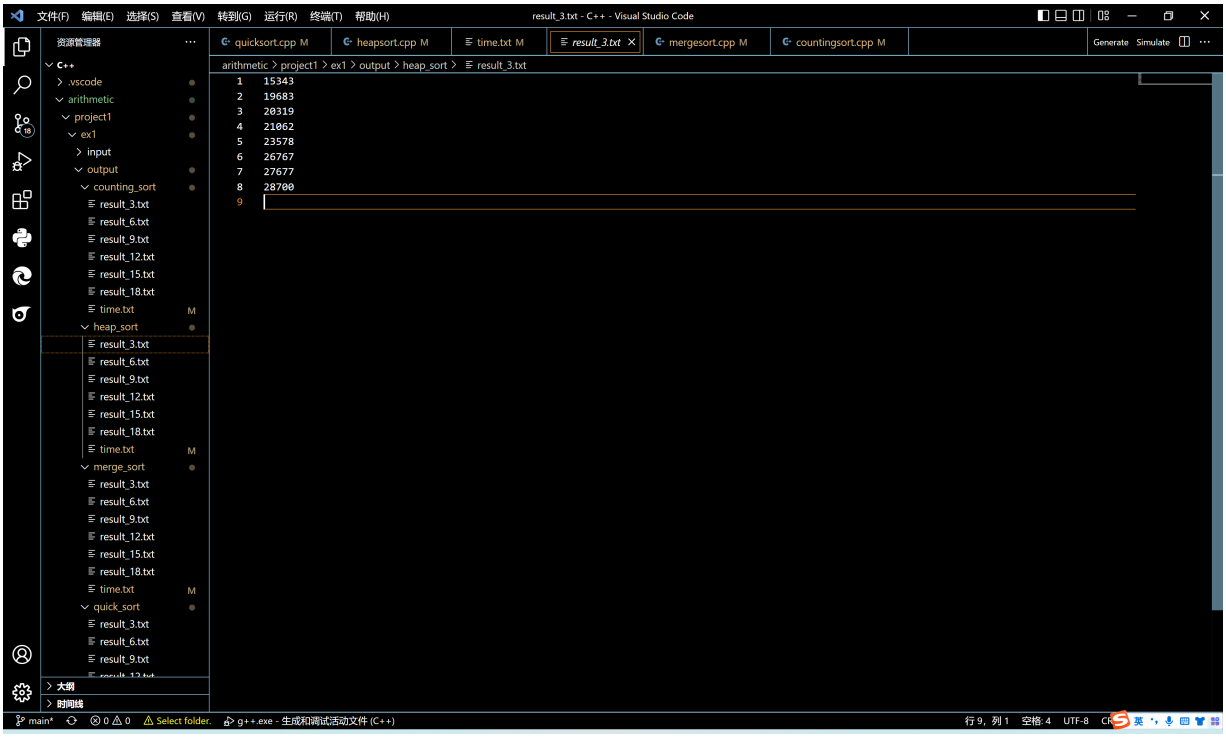
其中，待排序数组为arr，排好序的数组为sorted_arr。

2. **生成随机输入文件**
   - 使用cstdlib库的rand函数，生成2^{18}个范围为0-32767的随机数，存入input.txt
3. **运行代码并分析结果**
   - 在2^{18}的规模下，本机下如果不使用全局变量，会出现数组长度不够导致输入信息错误的情况，因此我将数组改为全局变量。
   - 每次实验我都自动使用循环来创建出所有规模的result和time输出文件，具体代码可以参考每个排序算法的的main函数。
   - 每个算法我都使用了5000次循环来累计时间，累计5000次后用clock输出秒数
   - 快速排序中的栈有可能达到2^{18}的长度，如果调用stack库中的栈，**栈的初始长度较小，可能会涉及到栈延长操作**，这会导致栈中数据大量复制，不能准确测出算法时间。因此，**我给自行实现的栈添加了用长度进行构造的构造函数。**
   - 对每个算法进行3次重复试验，算出平均时间，并用excel绘制图表

实验结果与分析

运行结果与时间截图

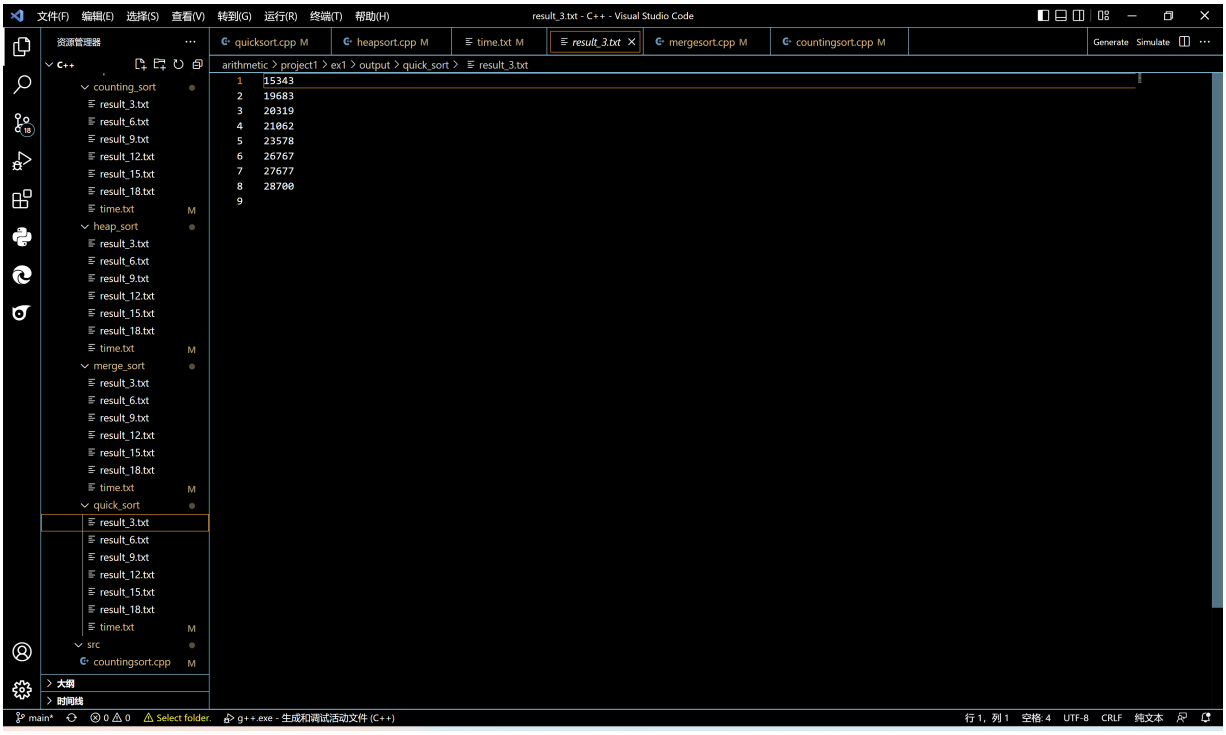
- 堆排序：
  - 运行结果



  - 运行时间

- 归并排序：
  - 运行结果

- 快速排序：
  - 运行结果



- 计数排序：
  - 运行结果

实验分析

四个排序的运行时间统计如下，为了更真实地反映排序的性能，我增加了2^{19}，2^{20}，2^{21}，三种输入规模：

- 堆排序

heapsort					
规模n	nlogn	运行时间time/ms			平均时间t/ms
		1	2	3	
8	24	0.0002	0.0002	0.0002	0.0002
64	384	0.003	0.0028	0.0022	0.002666667
512	4608	0.0316	0.0346	0.0312	0.032466667
4096	49152	0.5804	0.5506	0.5478	0.5596
32768	491520	5.6674	5.3472	5.3146	5.443066667
262144	4718592	57.114	53.0962	55.2486	55.15293333
524288	9961472	114	113.1882	112.9872	113.3918
1048576	20971520	260.8214	260.2016	260.1082	260.3770667
2097152	44040192	588.2212	587.9872	588.2198	588.1427333

通过最后一列可以看到，平均运行时间基本与$nlogn$成正比，在平均情况下，堆排序的时间复杂度$\Theta(nlogn)$可以由实验数据验证。

- 归并排序

mergesort					
规模n	nlogn	运行时间time/ms			平均时间t/ms
		1	2	3	
8	24	0.0003	0.0004	0.0006	0.000433333
64	384	0.0064	0.006	0.0086	0.007
512	4608	0.0628	0.063	0.0712	0.065666667
4096	49152	0.7726	0.7508	0.7548	0.7594
32768	491520	6.8688	6.846	6.8594	6.858066667
262144	4718592	65.1754	60.6534	60.6976	62.17546667
524288	9961472	127.8	127.8122	126.9824	127.5315333
1048576	20971520	264.2	263.9876	264.0824	264.09
2097152	44040192	529.2	529.2064	529.1874	529.1979333

最后一列依然显示出，归并排序的时间复杂度基本为$\Theta(nlogn)$。

- 快速排序

quicksort					
规模n	nlogn	运行时间time/ms			平均时间t/ms
		1	2	3	
8	24	0.0003	0.0002	0.0003	0.000266667
64	384	0.004	0.0048	0.0052	0.004666667
512	4608	0.0418	0.0416	0.0416	0.041666667
4096	49152	0.416	0.434	0.4248	0.424933333
32768	491520	4.6848	4.158	4.1156	4.319466667
262144	4718592	46.9786	42.8764	45.577	45.144
524288	9961472	85.122	88.124	86.224	86.49
1048576	20971520	206.2152	206.2	205.8675	206.0942333
2097152	44040192	480.2224	480.2164	480.9824	480.4737333

快速排序对于$\Theta(nlogn)$的时间复杂度契合的更好。由于我们每次排序需要运行5000次，虽然快速排序最坏情况下可能为$\Theta(n^2)$，但由于我采用了随机数优化算法，每次测量结果基本接近平均时间复杂度。

在规模很大时，快速排序的性能会逐渐降低，对归并排序的领先优势也不那么明显。后文我们详细讨论

- 计数排序

countingsort						
规模n	值域k	n+k	运行时间time/ms			平均时间t/ms
			1	2	3	
8	32768	32776	0.1852	0.1858	0.1798	0.1836
64	32768	32832	0.1918	0.1866	0.1858	0.188066667
512	32768	33280	0.1936	0.1758	0.177	0.182133333
4096	32768	36864	0.1978	0.1854	0.2002	0.194466667
32768	32768	65536	0.4034	0.3762	0.3852	0.388266667
262144	32768	294912	2.002	1.9468	1.948	1.9656
524288	32768	557056	5.4012	5.0098	5.6112	5.340733333
1048576	32768	1081344	12.8124	12.7342	12.6321	12.72623333
2097152	32768	2129920	28.8752	26.2018	27.1	27.39233333

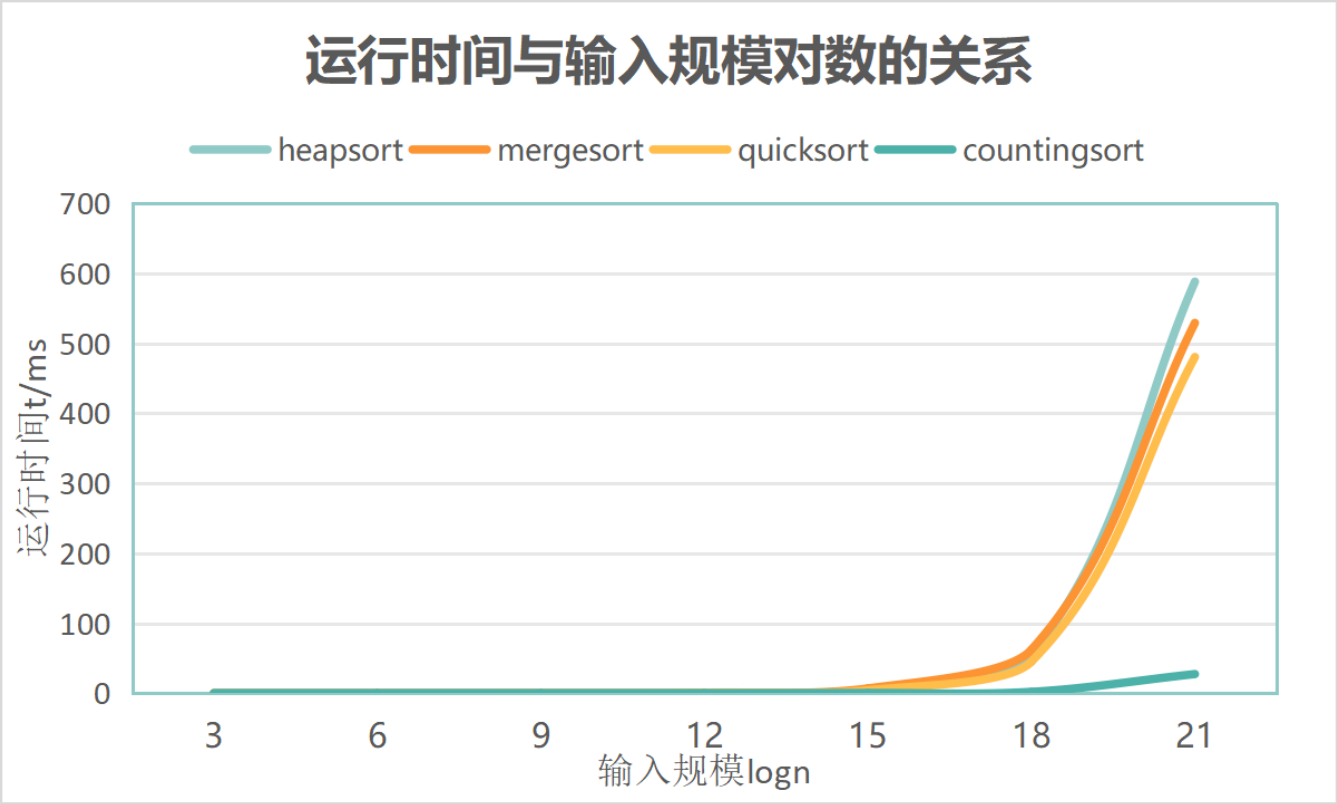
计数排序在输入规模n较小时基本保持了一致的运行时间，这是因为当n远小于k时，运行时间主要由k决定。但当输入规模达到32768以上时，n和k就一同决定了运行时间。但总体而言，计数排序的时间复杂度基本为$\Theta(n+k)$。

但是，计数排序在输入规模较小时有明显的运行时间反常现象。问题主要出现在**countingsort的count数组初始化和前缀和计算**里

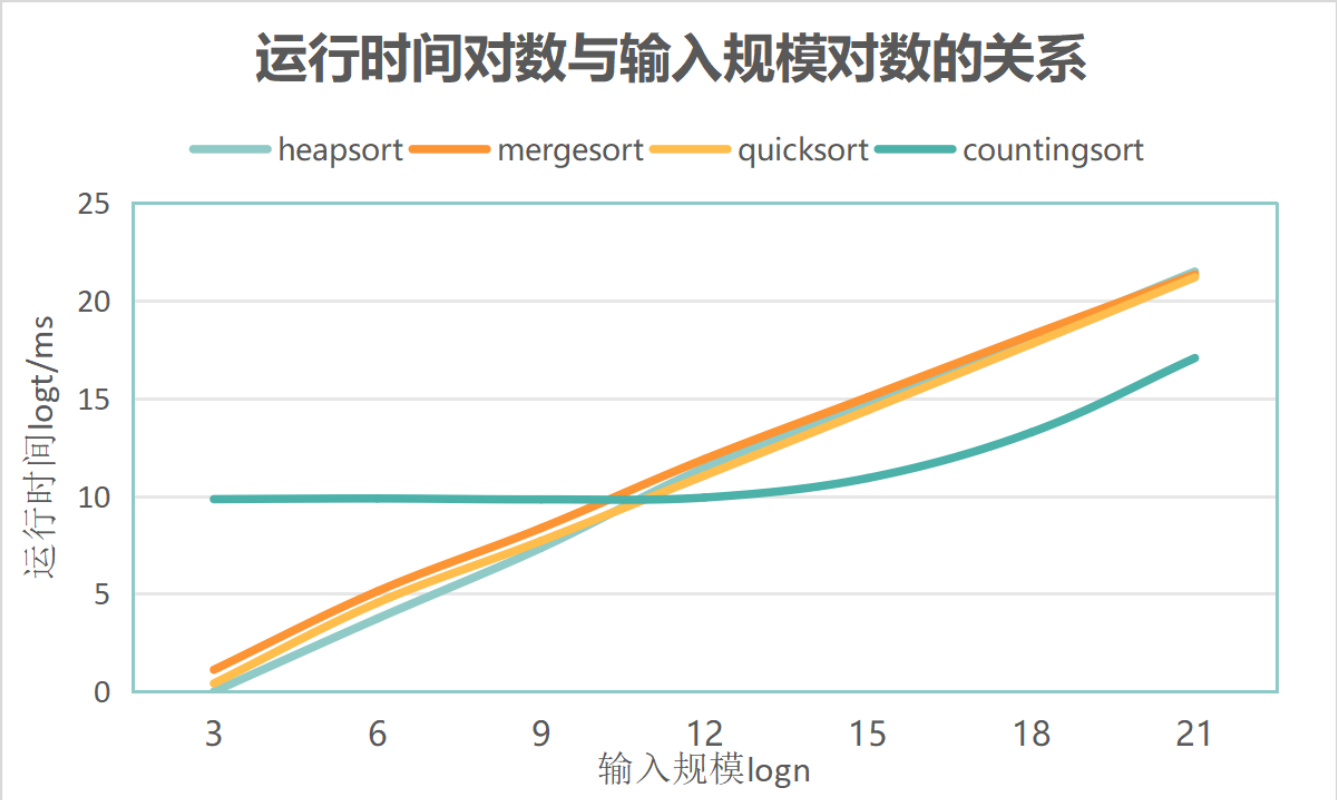
- 当n比较小时，32768次count数组的访存操作显然耗费了太多的时间，导致其他操作的时间完全可以忽视
- 对于输入规模为512和64时的反常现象，我认为很有可能是高速缓存的原因。在输入规模较小时，高速缓存前期的缺失是无法避免的，而缺失处理的时间的波动本身就比较可观，加之32768次访存时间确实较长，因此造成了反常现象。

在输入规模较大时，受到缺页处理、高速缓存的影响，计数排序虽然基本保证了$\Theta(n+k)$的时间复杂度，但时间常数也会不断增大，导致了运行时间的逐步加长。

下面我们对四个排序进行比较：



在求两个对数之间的关系图表时，为了保证以2为底的时间对数为正数，我对求对数后的时间统一加了一个常数：



- 在输入规模较大时，参考"运行时间与输入规模对数的关系"图表。
  - 若可以得知数据的范围，计数排序有着显著的优势，它永远比其他基于比较的排序时间少一个量级。
  - 若未知数据范围，我们可以看到，**当规模小于2^{20}次方时，快速排序有着较为显著的优势**。但当规模更大时，由于我使用C++来实现了几个排序算法，**C++对于比较的优化是很大的**，这导致了虽然快速排序比较次数少于归并排序，但快速排序的栈操作和分割操作涉及了更多的数据移动，导致了快速排序的性能略微降低。而**归并排序由于需要相当稳定地进行逐层归并，因此时间与规模的增长比例也基本稳定**。所以，由于我自己实现的快速排序空间占用量高，可能造成了一定的性能降低。
- 在输入规模较小时，参考"运行时间对数与输入规模对数的关系"图表。

  这时，计数排序因为数据规模较大，已经失去了大规模时的优势。而堆排序在此时却很有优势。由于**MAX_HEAPIFY函数有中途退出（break）机制**，当输入规模在512以下时，大根堆的调整有很大概率中途完成，而归并排序必须要严格按照数组长度进行合并，因此堆排序性能较好。但随着输入规模较大，数组最后一个元素是较大元的概率也逐渐降低，因此每次将其换到根节点时，基本上都需要向下移动一个满树高的距离，故性能逐渐降低，而快速排序在此时性能优势逐渐体现了。
- 综上所述：
  - 在数据规模较小时，可以使用堆排序，获得更好的时间性能，或者使用归并排序，获得更稳定的时间性能
  - 在数据规模较大时，使用快速排序可以获得更好的时间性能。若已知数据范围，则可以使用计数排序，时间性能更好。

总结与心得

经过本次实验，我有了以下收获：

1. 对各类排序算法有了更加深刻的理解，并可以自行实现各类排序
2. 对快速排序的较好的时间性能进行了分析，充分理解了快速排序其名称中的"快速"二字
3. 了解了excel绘图功能的强大，学会了使用excel绘制出精美的图表