**深 圳 大 学 实 验 报 告**

**课程名称： 算法设计与分析**

**实验名称： 排序算法性能分析**

**学院： 计算机与软件学院 专业： 计算机科学与技术**

**报告人： 钟海旋 学号： 2023150140 班级： 计算机科学与技术高性能班**

**同组人： 无**

**指导教师： 杨烜**

**实验时间： 2025年2月28日-2025年3月21日**

**实验报告提交时间： 2025年3月21日星期五**

**教务处制**

**一．实验目的**

* 掌握选择排序、冒泡排序、插入排序、合并排序、快速排序算法原理
* 掌握不同排序算法时间效率的经验分析方法，验证理论分析与经验分析的一致性。
* 求解TOP K问题，并分析比较不同算法效率。

**三．实验概述**

排序问题要求我们按照升序排列给定列表中的数据项，目前为止，已有多种排序算法提出。本实验要求掌握选择排序、冒泡排序、插入排序、合并排序、快速排序算法原理，并进行代码实现。通过对大量样本的测试结果，统计不同排序算法的时间效率与输入规模的关系，通过经验分析方法，展示不同排序算法的时间复杂度，并与理论分析的基本运算次数做比较，验证理论分析结论的正确性。

**二．实验步骤与结果**

1、实现选择排序、冒泡排序、插入排序、合并排序、快速排序算法；

2、以待排序数组的大小n为输入规模，固定n，随机产生20组测试样本，统计不同排序算法在20个样本上的平均运行时间；

3、分别以n=10万, n=20万, n=30万, n=40万, n=50万等等，重复2的实验，画出不同排序算法在20个随机样本的平均运行时间与输入规模n的关系，如下图1所示，注意横坐标要均匀。

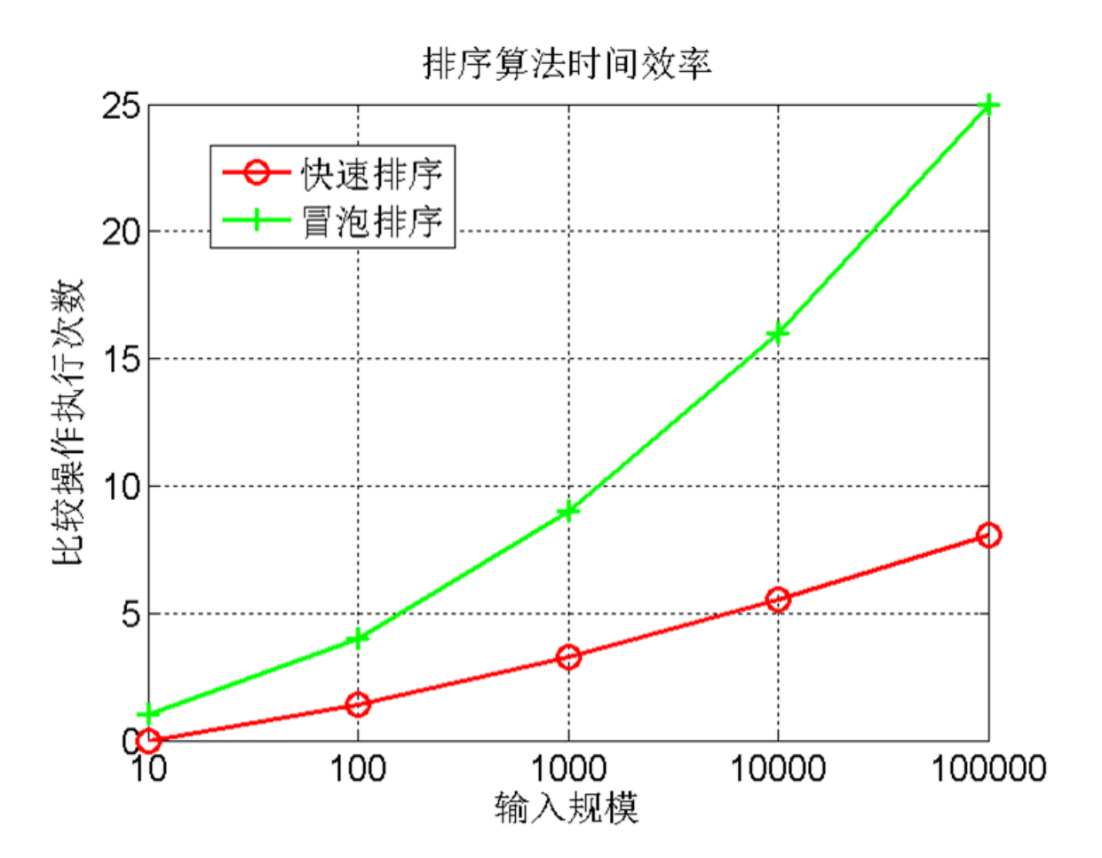


图1. 时间效率与输入规模n的关系图

4、画出理论效率分析的曲线和实测的效率曲线，注意：由于实测效率是运行时间，而理论效率是基本操作的执行次数，两者需要进行对应关系调整。调整思路举例：以输入规模为10万的数据运行时间为基准点，计算输入规模为其他值的理论运行时间，画出不同规模数据的理论运行时间曲线，并与实测的效率曲线进行比较。经验分析与理论分析是否一致？如果不一致，请解释存在的原因。

5、现在有10亿的数据（每个数据四个字节），请快速挑选出最大的十个数，并在小规模数据上验证算法的正确性。

**三．实验分析**

* **冒泡排序**

1. **算法原理描述：**

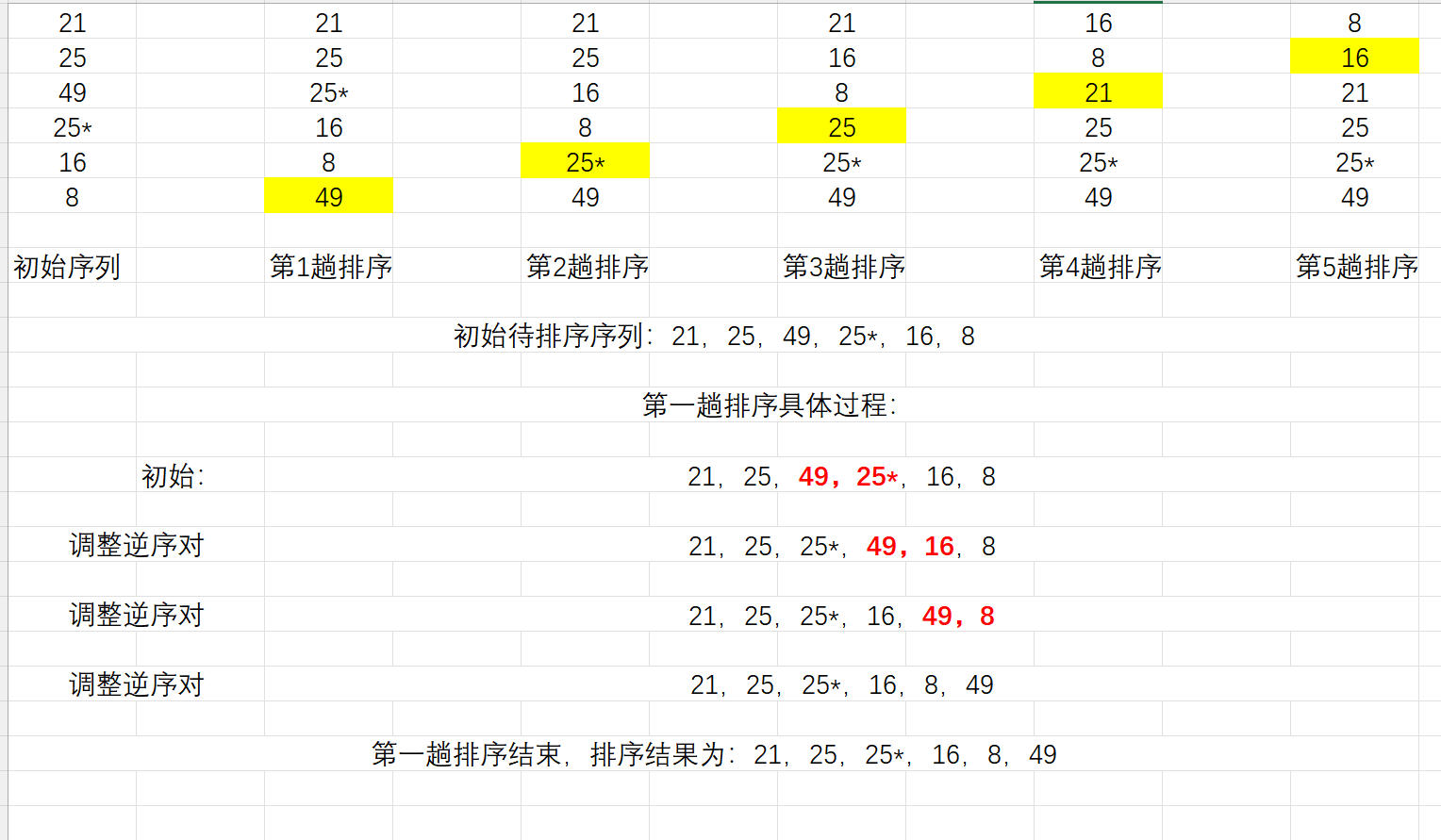
对于n个待排序记录，从第1个记录起，依次比较相邻两个记录的关键字，如果发生逆序，则交换之，直至第n-1个记录和第n个记录比较完，此时最后一个记录是数值最高的记录；以此类推，下一次待排序的序列是第一个记录到第n-1个记录的序列，重复第一次的操作，一轮结果之后最后一个（n-1）记录是次高的记录；重复操作，直至序列全部有序。

1. **简述：**

重复地遍历待排序序列，比较相邻元素，如果顺序有误则交换它们的位置，每一轮遍历结束后，最大的元素会“冒泡”到序列的末尾。通过不断减少待排序序列的长度，最终得到完全有序的序列。

1. **图解：**

* 动图：[d0be3f0d2919760609dd9a5d824639a5.gif (893×407)](https://i-blog.csdnimg.cn/blog_migrate/d0be3f0d2919760609dd9a5d824639a5.gif)
* 静态图图解：



1. **核心伪代码：**

* 优化前：

Bubblesort(array,n) // n表示array数组的长度

for i in range(n - 1): //两层循环

for j in range(n - i - 1):

if array[j] > array[j + 1]: //出现逆序对，交换 array[j]和array[j + 1]

temp = array[j]

array[j] = array[j + 1]

array[j + 1] = temp

* 优化后：

Bubblesort(array,n) // n表示array数组的长度

for i in range(n - 1):

flag=flase //设置标志位flag

for j in range(n - i - 1):

if array[j] > array[j + 1]: //出现逆序对交换array[j] 和 array[j + 1]

temp = array[j]

array[j] = array[j + 1]

array[j + 1] = temp

flag=true //标志位flag设为true，说明有交换过

if flag ==flase: //flag=false->目前序列里无逆序对，有序

break //已经全部有序，退出循环

1. **算法测试结果及效率分析**

* 优化前：

1. 时间复杂度：O(n2)
2. 空间复杂度：辅助变量O(1)，因为只使用了常数级别的额外空间
3. 稳定性：稳定的，因为在交换时，只有当前面的元素大于后面的元素才进行交换，不会改变相同元素的相对顺序。

* 优化后：

1. 时间复杂度：

在最坏情况下（即数组完全逆序）：O(n2)

在最佳情况下（即数组已经有序）：O(n)

1. 空间复杂度：辅助变量O(1)，因为只使用了常数级别的额外空间
2. 稳定性：仍然稳定，交换条件没有发生变化

* 测试数据图表：

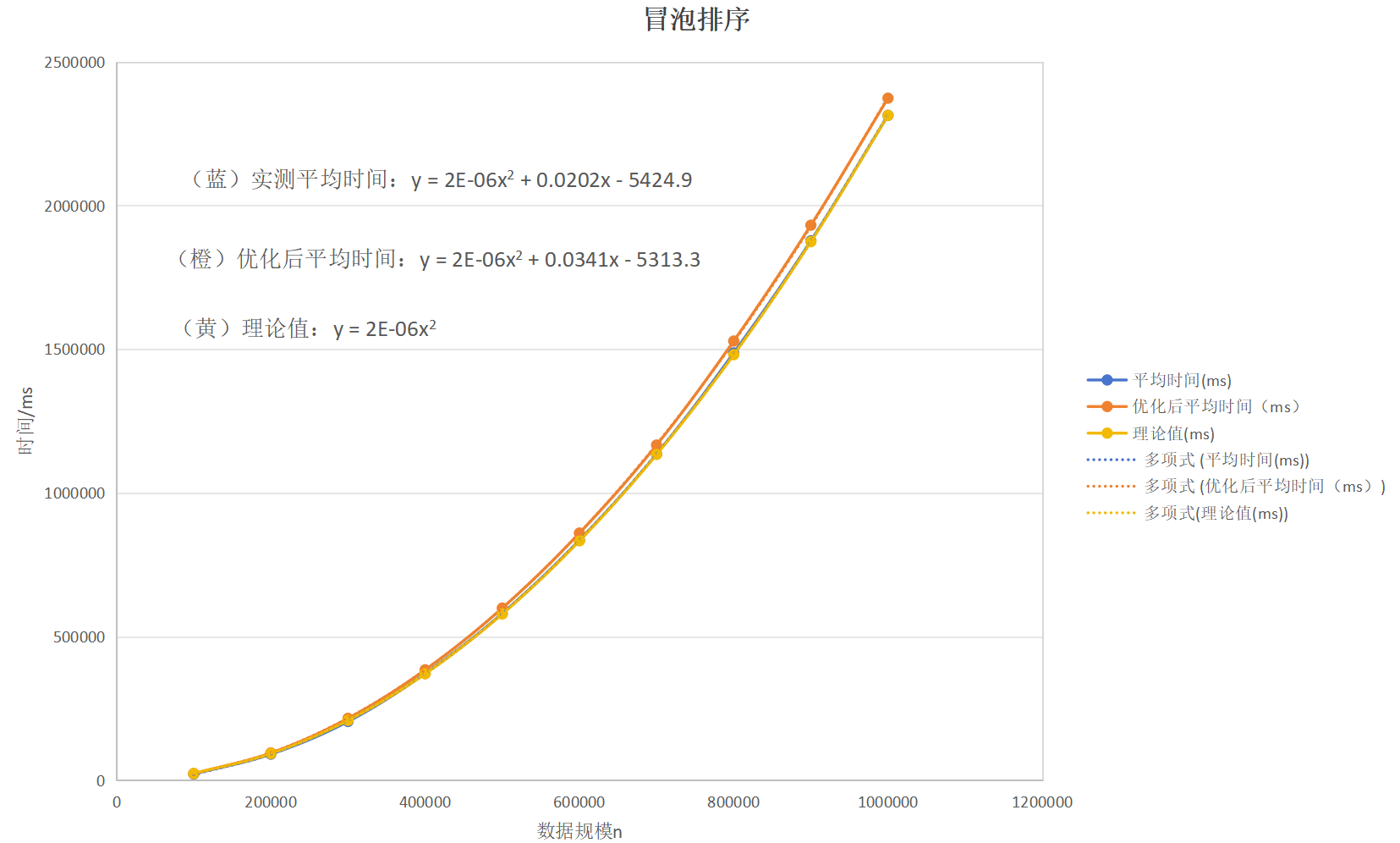
理论值计算：

根据冒泡排序算法的时间复杂度 O(n2)，并以100万数据规模的理论时间值 T100万=2314030ms 为基准，公式：Tn=T100万×( )2  。

注意：这里的理论值都是没有优化的算法的理论值



对应图像：



* 分析：

1. 优化前后对比：

理论上来说，优化后的版本常常能带来明显的性能提升，特别是在数据部分有序的情况下。但是我们观察图表数据，发现，优化后的冒泡排序时间反而变慢了，猜测这可能与数据分布有关，数据随机生成混乱程度太大，而且有多了判断条件，故而增加了执行时间。

1. 理论和实测对比

理论值与实际测量值比较接近，尤其是在数据规模较大时，两者的差距较小。实际执行时间与理论值较为接近，验证了冒泡排序的时间复杂度模型：O（n2）

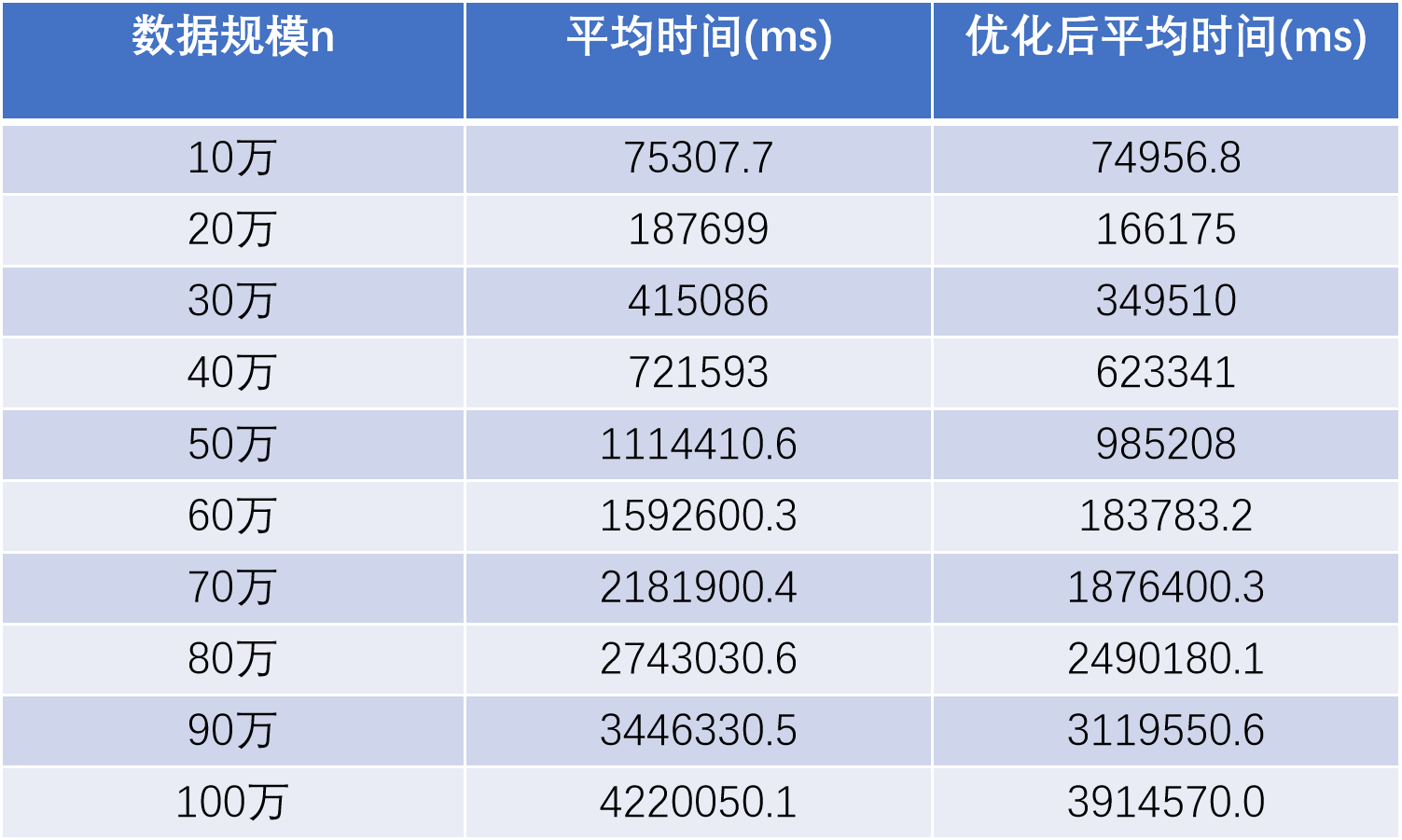
1. 数据规模与平均时间

随着数据规模的增加，冒泡排序的平均执行时间显著增加。

1. 性能分析

冒泡排序在处理大规模数据时性能较差（O（n2）），随着数据规模的增加，执行时间会显著增加

1. 优化前后误差原因验证：



分析表格：

构造一个**相对有序**的数据集，测试优化前后运行时间，我发现，此时优化后的运行时间确实变短了，说明在一个相对有序的数据集里面，这个优化方案是可行的。

结论：验证了之前在数据高度无序（随机生成）的情况下，优化前后的误差出现的猜测！

* **选择排序**

1. **算法原理描述：**

（升序）对于有n个记录的待排序的序列，第一趟从第一个记录开始寻找最小的记录的下标，从而将它与第一个记录交换位置；接着从第2个记录到第n个记录就是下一趟的n-1个待排序的序列，重复第一趟的操作，找到这个待排序的序列的最小的记录的下标，将它与该待排序序列的第一个记录交换位置，此时我们已经找到了最小的两个记录并排序；一次循环往复直至序列全部有序。

1. **简述：**

不断地选择剩余序列中的最小记录，并将其放到已排序序列的末尾，从而逐步完成整个序列的排序

1. **图解：**

* 动图**：**[**d32bbdf44bf0f2ff92ee3abfcac0a450.gif (836×610)**](https://i-blog.csdnimg.cn/blog_migrate/d32bbdf44bf0f2ff92ee3abfcac0a450.gif)
* 静态图图解：



1. **核心伪代码：**

* 优化前：

Select\_sort(array，n):

for i in range(n): //假设当前i位置是最小值的索引

min\_index = i //从i+1位置开始遍历到数组末尾

for j in range(i + 1, n):

if array[min\_index] > array[j]: //发现更小的值，更新最小值的索引

min\_index = j

Swap(array[min\_index]，array[i] ) //将找到的最小值与当前i位置的值进行交换

* 优化后：（双向选择排序）

Optimize\_Selectsort(array,n)

for i: 0 ->n-1 , j: n-1->0 , && i<j

minIndex=i //记录最小值索引

maxIndex=i //记录最大值索引

for k : i -> j

if array[k]<array[minIndex] //找到更小数据，更新最小值索引

minIndex=k

else if array[k]>array[maxIndex] //找到更大数据，更新最大值索引

maxIndex=k

swap(array[i],array[minIndex]); //交换位置

swap(array[j],array[maxIndex]);

优化思路：在找最小值的时候顺便把最大值也找了，这样可以减少循环的遍历次数

1. **算法测试结果及效率分析**

* 优化前：

1. 时间复杂度：O(n2)
2. 空间复杂度：辅助变量O(1)，因为只使用了常数级别的额外空间
3. 稳定性：不稳定的，因为在交换过程中可能会改变相同元素的相对顺序， eg:5,3,6,9,8,2,5\*,1排序后：1,2,3,5\*,5,6,8,9

* 优化后：

1. 时间复杂度：

实际运行中，可能会减少大约一半的比较次数，但整体复杂度不变，还是O(n2)

【分析：n-2+n-4+…+1(或者0，具体取决于n为奇数或者偶数)≈(1+n-2)\*n/2/2=(n-1)\*n/4】

1. 空间复杂度：辅助变量O(1)，因为只使用了常数级别的额外空间
2. 稳定性：不稳定的，因为在交换过程中可能会改变相同元素的相对顺序

* 测试数据图表：

理论值计算：

根据选择排序算法的时间复杂度 O(n2)，并以30万数据规模的理论时间值 T30万=60572.6 ms 为基准，公式：Tn=T30万×( )2  。

注意：这里的理论值都是没有优化的算法的理论值

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **数据规模n** | **平均时间(ms)** | **平均时间对数** | **优化后平均时间（ms）** | **优化后平均时间对数** | **理论值(ms)** | **理论值对数** |
| 10万 | 6683.53 | 3.825006 | 5512.18 | 3.741323 | 6730.3 | 3.828034 |
| 20万 | 27234.1 | 4.435113 | 22003.3 | 4.342488 | 26921.2 | 4.430094 |
| 30万 | 60572.6 | 4.782276 | 49325.5 | 4.693071 | 60572.6 | 4.782276 |
| 40万 | 107624 | 5.031909 | 88516.1 | 4.947022 | 107684.6 | 5.032154 |
| 50万 | 168575 | 5.226793 | 137755 | 5.139107 | 168255.6 | 5.22597 |
| 60万 | 242522 | 5.384751 | 198907 | 5.29865 | 242290.4 | 5.384336 |
| 70万 | 328147 | 5.516068 | 268502 | 5.428948 | 329784.2 | 5.51823 |
| 80万 | 422951 | 5.62629 | 352109 | 5.546677 | 430738.5 | 5.634214 |
| 90万 | 529014 | 5.723467 | 444844 | 5.648208 | 545153.4 | 5.736519 |
| 100万 | 660859 | 5.820109 | 545384 | 5.736702 | 673029.0 | 5.828034 |

图像：

* **分析**：

1. 效率评估

选择排序在处理小规模数据时可能表现尚可，但由于其 O(n2)的时间复杂度，随着数据规模的增大，运行时间增长得非常快。这使得它在处理大规模数据时效率较低

1. 理论和实测对比

理论值与实际测量值比较接近,实际执行时间与理论值较为接近，两条曲线几乎重合，验证了选择排序的时间复杂度模型：O（n2）

1. 数据规模与平均时间

随着数据规模的增加，冒泡排序的平均执行时间显著增加

* **插入排序**

1. **算法原理描述：**

将待排序的记录Ri，插入到已排好序的记录表R1, R2 ,…, Ri-1中，得到一个新的、记录数增加1的有序表，重复操作直到所有的记录都插入完为止。

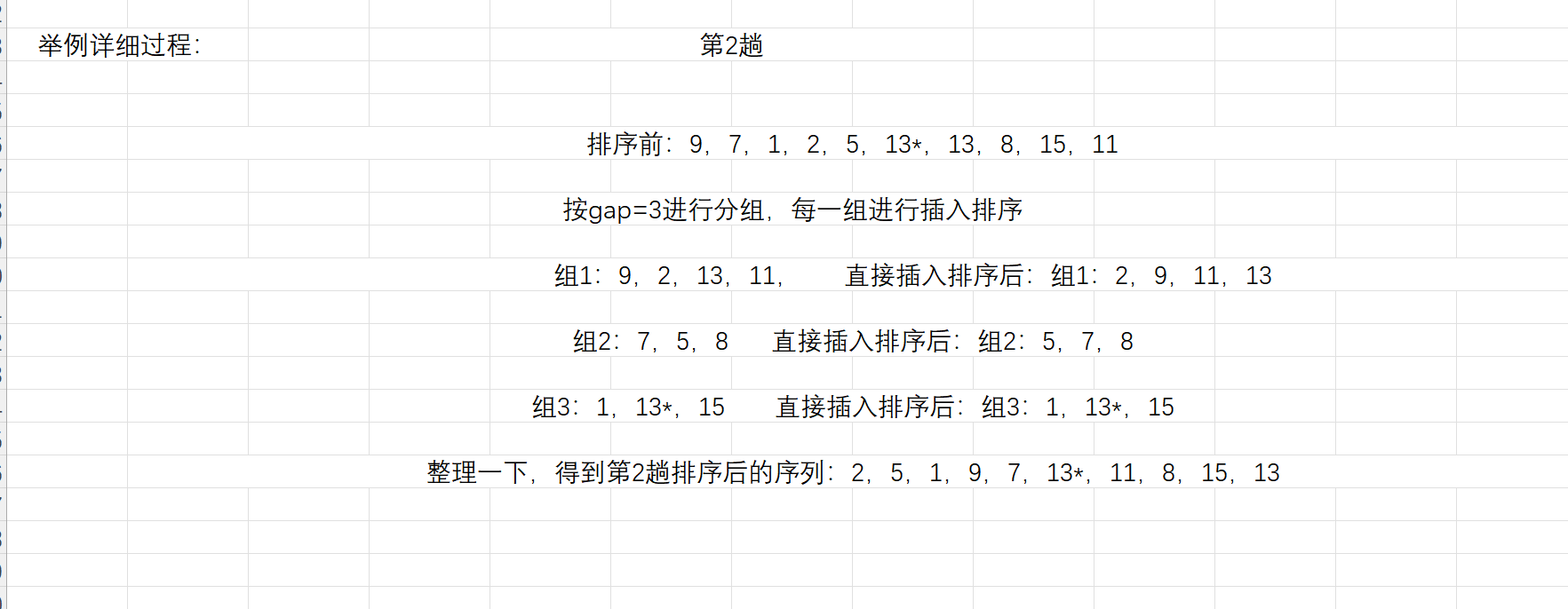
1. **图解：**

* **动图：**[**577d3d2efc3aa392f****e028dc7ae05ae6f.gif (804×1073)**](https://i-blog.csdnimg.cn/blog_migrate/577d3d2efc3aa392fe028dc7ae05ae6f.gif)
* **静图：**直接插入排序如下图所示：









1. **核心伪代码：**

* 优化前：直接插入排序

Insertsort(int array[],int n):

for i =1 to n //从第二个元素开始第一个元素，默认是已排序的

temp = array[i] //暂存当前要插入的元素

j = i - 1

while j >= 0 and array[j] > temp: //在已排序部分从后向前扫描，找合适位置插入

array[j + 1] = array[j] // 当前扫描元素大于要插入的元素，向后移动

j -= 1

array[j + 1] = temp // 找到合适的位置，将暂存的元素插入

* 优化后：希尔排序

Optimize\_insertsort(int array[],int n):

gap = n/ 2 //缩小增量, 数组长度一半作为初始间隔

while gap > 0:

tmp=0

for i in range(gap, n): //从gap索引开始遍历数组

temp = array[i]

j = i

while j >= gap && array[j - gap] < temp:

array[j] = array[j - gap] //对当前元素间隔gap的元素进行直插排序 j -= gap

array[j] = temp

gap =gap/2 //缩小增量，除二

1. **算法测试结果及效率分析**

* 优化前：直接插入排序

1. 时间复杂度：

* 最坏和平均情况下的时间复杂度为 O(n2)
* 最好情况下时间复杂度为O(n)

1. 空间复杂度：辅助变量O(1)，因为只使用了常数级别的额外空间
2. 稳定性：稳定的，因为相等元素在排序后仍保持相对顺序

* 优化后：希尔排序

1. 时间复杂度：取决于步长序列的选择。通常介于 O(nlog2n) 和 O(n2) 之间
2. 空间复杂度：辅助变量O(1)，因为只使用了常数级别的额外空间
3. 稳定性：不稳定的，因为步长变化可能导致相等元素的相对顺序改变

* 测试数据图表：

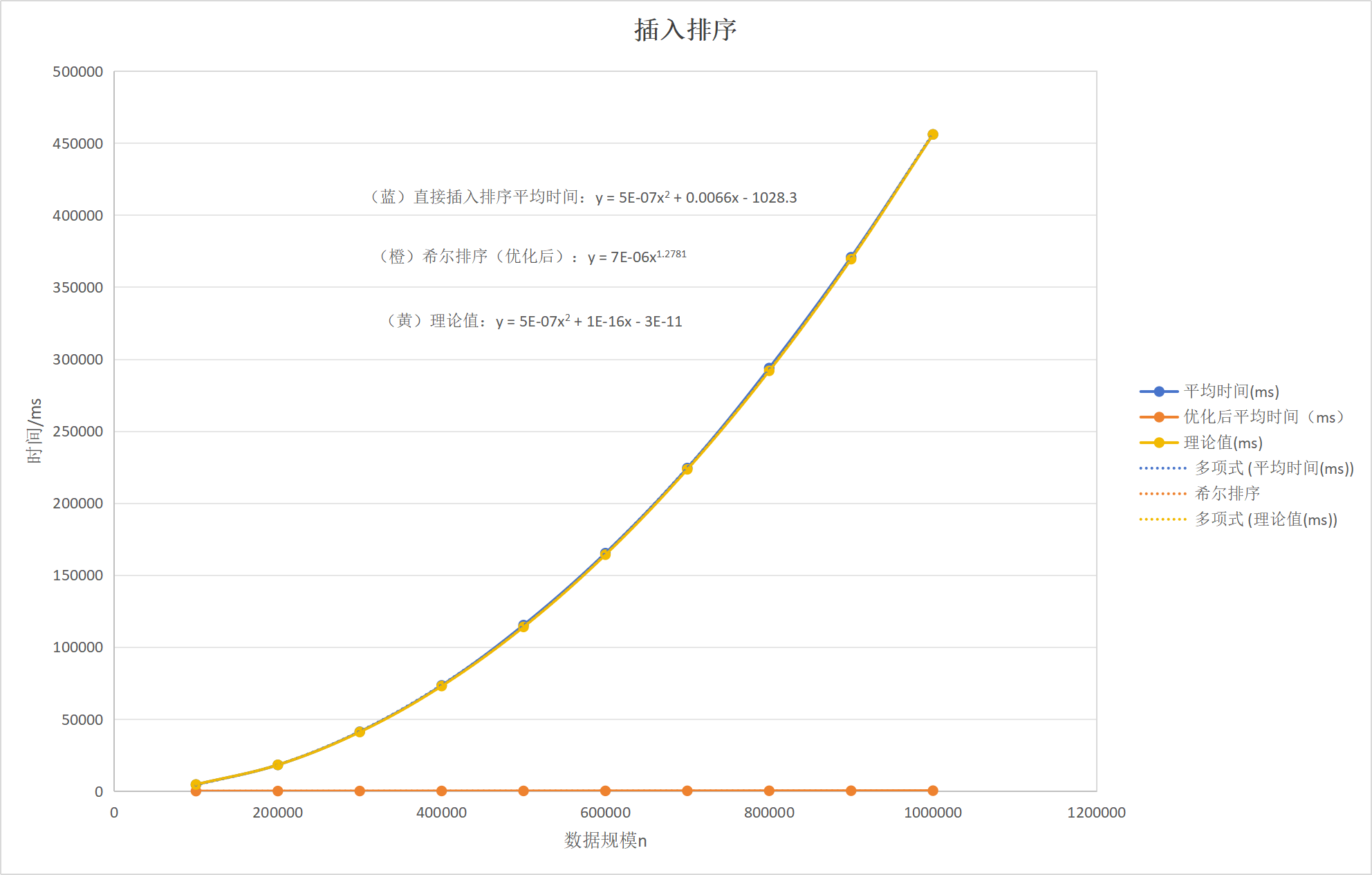
理论值计算：

根据直接插入排序算法的时间复杂度 O(n2)，并以30万数据规模的理论时间值 T100万=455965ms 为基准，公式：Tn=T100万×( )2  。

注意：这里的理论值都是没有优化的算法的理论值



* 图像：



* 分析：

1. 优化前后对比：

很明显，优化后的插入排序（希尔排序）性能得到了大幅度提升。相比于直接插入排序，希尔排序在稍大规模的数据上更具优势。

1. 理论和实测对比

理论值与实际测量值比较接近，尤其是在数据规模较大时，两者的差距较小。实际执行时间与理论值较为接近，验证了直接插入排序的时间复杂度模型：O（n2）

1. 数据规模与平均时间

随着数据规模的增加，直接插入排序的平均执行时间显著增加。

1. 性能分析

直接插入排序在处理大规模数据时性能较差（O（n2）），随着数据规模的增加，执行时间会显著增加。因此，可以得出结论：直接插入排序适用于小规模或基本有序的数据集以及希望得到的相等数据相对顺序不发生变化，而希尔排序适用于对稳定性没有要求、大规模数据集。

* **归并排序**

1. 算法原理描述：
2. 初始划分：

将待排序的数组视为由 n 个长度为 1 的有序子序列组成。每个元素单独看作一个有序序列。

1. 归并过程：

不断将相邻的两个有序子序列归并为一个新的有序子序列。

归并操作的关键在于：给定两个有序的子序列，通过比较它们的元素，将它们合并为一个新的有序序列。

1. 迭代合并：

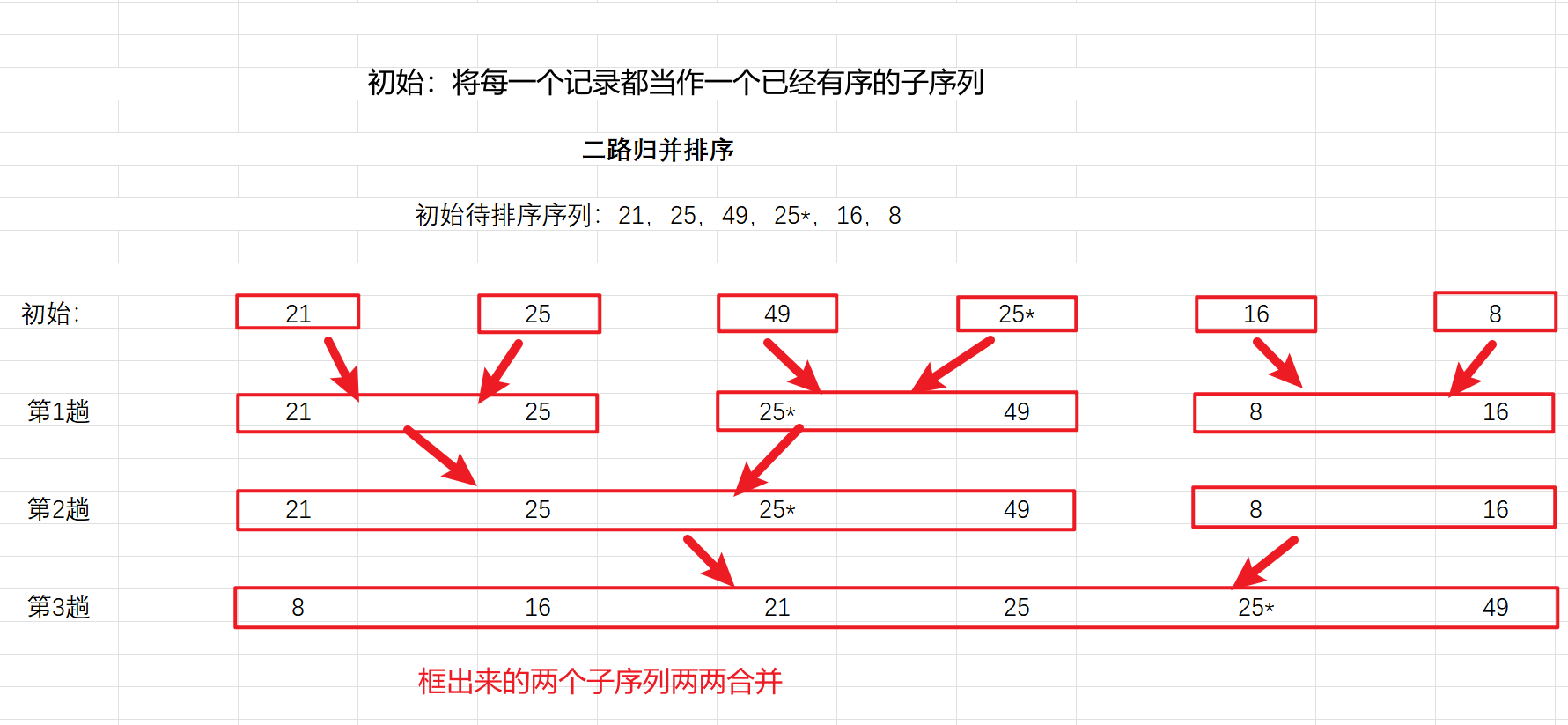
重复上述归并过程，逐步扩大有序子序列的长度。

每次合并后，有序子序列的数量减少，但每个子序列的长度增加。

最终，所有子序列合并为一个长度为 n 的完整有序序列。

1. 简述：将两个或两个以上的有序序列合并成一个有序序列
2. 图解：

* 动图：[665863fe42394b63c6b5fc32b8ec71ba.gif (1022×644)](https://i-blog.csdnimg.cn/blog_migrate/665863fe42394b63c6b5fc32b8ec71ba.gif)
* 静图**：**



1. 核心伪代码：

* 划分递归操作：

MergeSort(array, left, right)

if left<right //是否结束条件

mid=(left+right)/2 //对半划分

mergesort(array,left,mid)

mergesort(array,mid+1,right）

if array[mid]>array[mid+1] //优化：无序再合并

merge(array,left,mid,right) //合并两个有序子序列为一个序列

* 合并两个有序子序列：

Merge( array, left, mid, right)

i=left //第一个子序列的起始位置

j=mid+1 //第二个子序列的起始位置

int k=0;

temp[right-left+1] //临时数组

While i<=mid&&j<=right //到其中一个子序列末端跳出循环

If array[i]<=array[j] //合并两个子序列

temp[k++]=array[i++]

else

temp[k++]=array[j++]

while i<=mid //将剩余的元素放入临时数组

temp[k++]=array[i++]

while j<=right

temp[k++]=array[j++] //将临时数组元素复制到原数组中

for i=0 to k:

array[left+i]=temp[i]; //细节：从原数组的left开始插入，这里是合并的子序列的起始位置

1. **算法效率分析**

* 优化前：

1. 时间复杂度：

每一层递归的合并操作 O(n) +递归树的深度 log2 n=总时间复杂度为 O(n log n)。

1. 空间复杂度：

额外的空间来存储合并过程中的临时数组-O(n)

1. 稳定性：

稳定的，因为相等元素在排序后仍保持相对顺序

* 优化后：

时间复杂度、空间复杂度、稳定性不变，在部分有序的情况下，可以减少不必要的合并操作，提高效率。

* 测试数据图表：

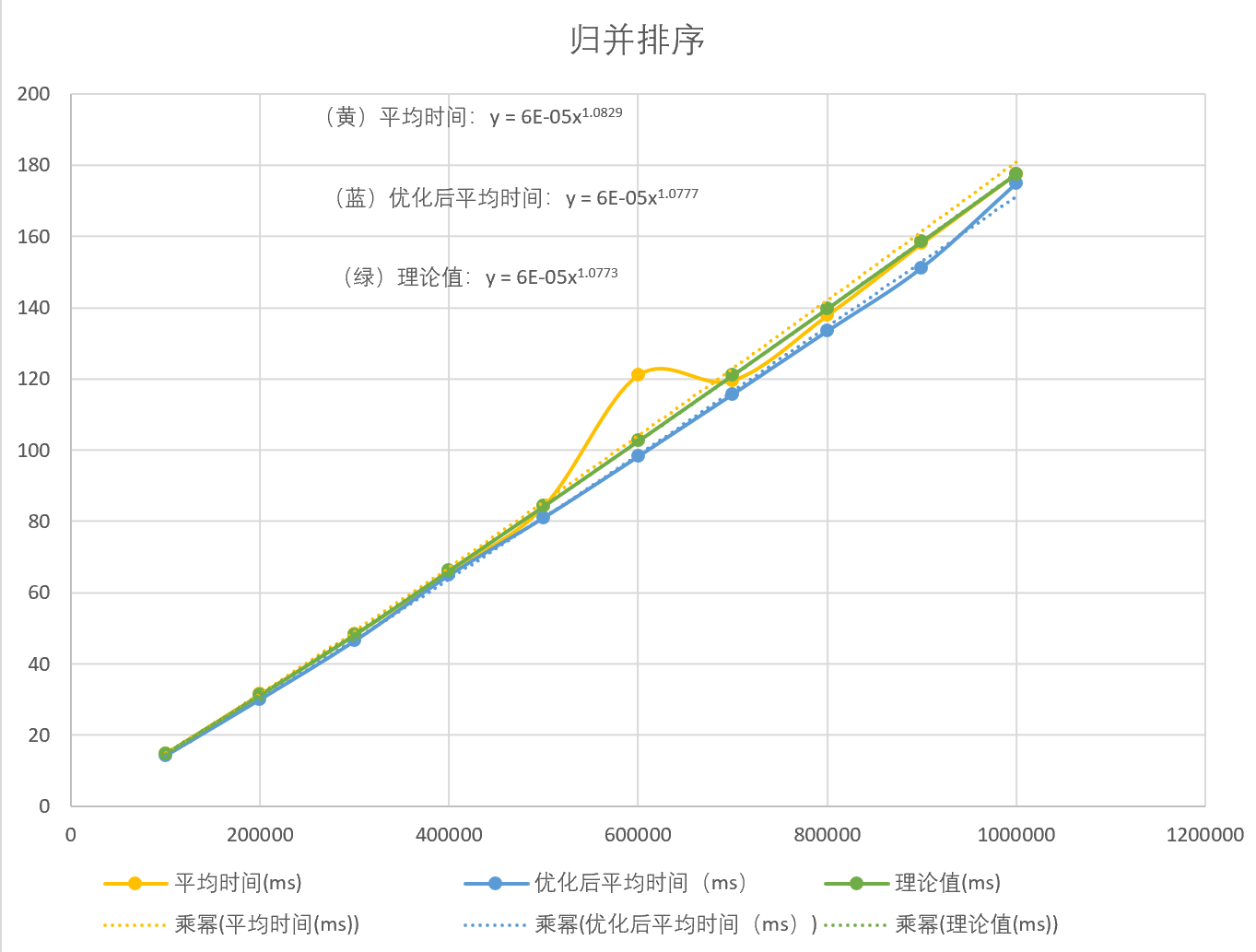
理论值计算：

根据归并排序算法的时间复杂度 O(n log n)，并以100万数据规模的理论时间值 T100万=177.557 ms 为基准，公式：Tn=T100万× 。

注意：这里的理论值都是没有优化的算法的理论值



* 图像：



* 分析：

1. 优化前后对比：

* 优化后的平均时间略低于未优化的平均时间，表明在某些情况下合并操作被成功跳过
* 优化效果在较小规模的数据上不太明显，因为数据接近无序的可能性更高

1. 理论和实测对比

理论值与实际测量值在较大数据规模下较为接近，表明归并排序的时间复杂度模型 O(nlogn) 得到了验证。尽管个别数据组因随机性导致偏差较大，但整体误差在合理范围内，实际执行时间与理论预测基本吻合。这进一步证实了归并排序的高效性和稳定性。

1. 数据规模与平均时间

随着数据规模 n 的增加，平均时间和优化后平均时间都呈现出对数增长的趋势，这符合归并排序的时间复杂度 O(nlogn)。

1. 性能分析

归并排序是一种稳定且适应性良好的排序算法，在处理较大规模的数据集仍速度较快，并且算法还是稳定的，优化后也取得良好的效果，可以说，效率和稳定性之间取得了良好的平衡

* **快速排序**

1. **算法原理描述**
2. 选择枢轴：

从待排序的序列中任选一个记录作为枢轴，可以是第一个。

1. 分区操作：

* 通过一趟排序，将待排序序列以枢轴为界分割成两部分，分区后，枢轴位于其最终排序位置。
* 分区规则：

枢轴前面的部分包含所有比枢轴关键字小的记录。

枢轴后面的部分包含所有比枢轴关键字大的记录。

1. 递归排序：

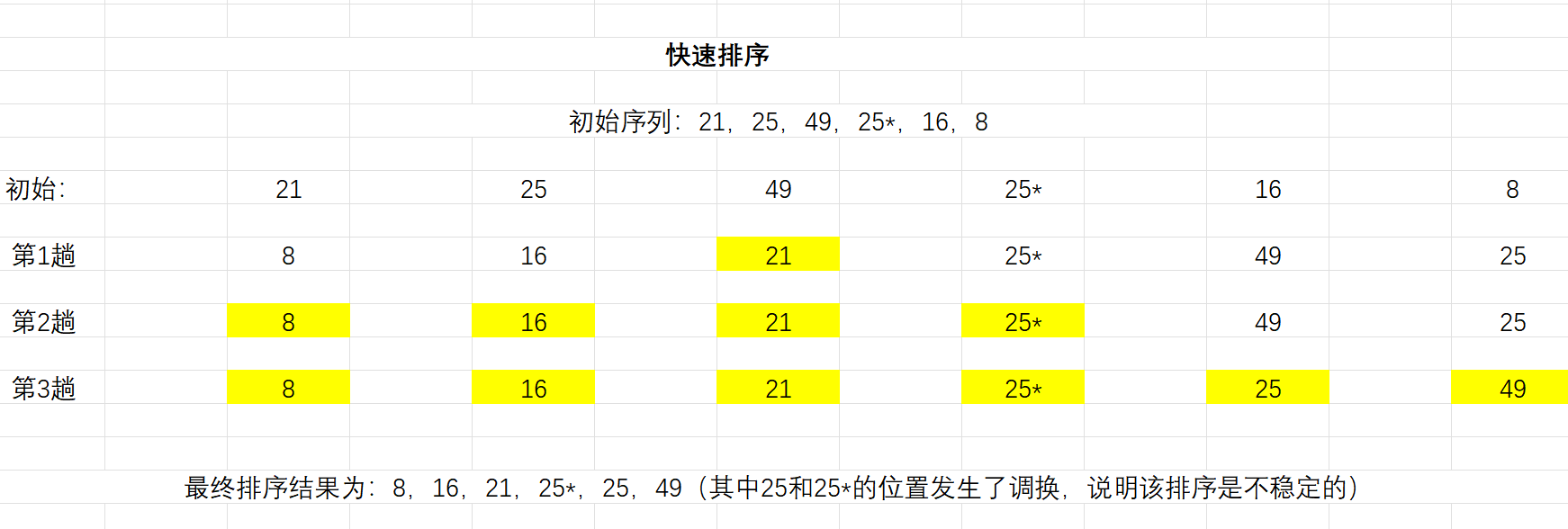
对枢轴前面的子序列和枢轴后面的子序列，分别采用同样的方法进行排序。递归地对每个子序列进行分区和排序，直到子序列的长度为1或0，此时序列自然有序。

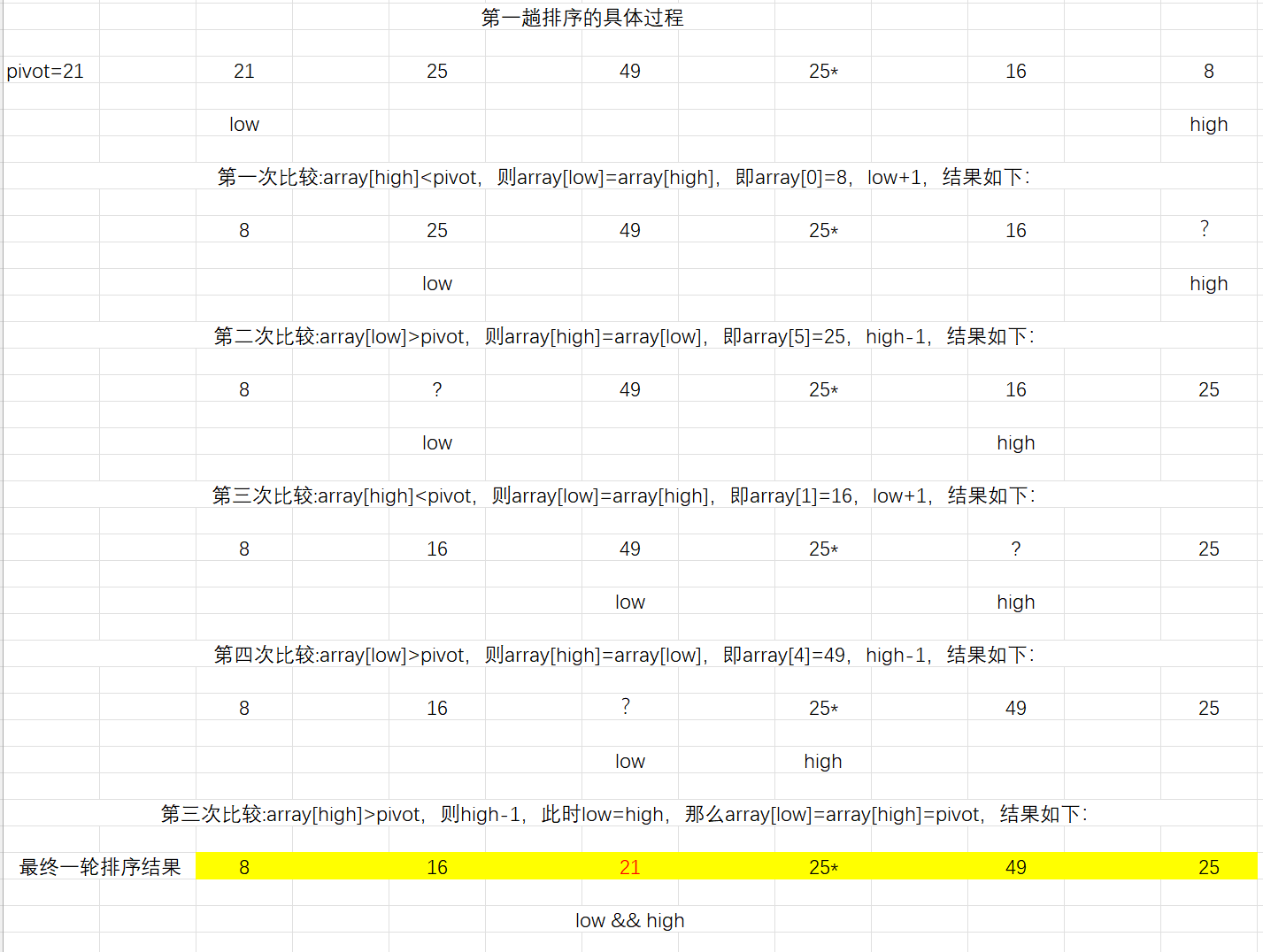
1. 合并结果：

由于每次分区操作已经将序列分为有序的两部分，且枢轴位于正确位置，整个序列在递归完成后即变为有序。

1. **核心思想：**分治策略，将大问题分解为小问题，通过解决小问题来解决大问题，最终达到整个序列有序的目的**。**
2. **图解：**

* **动图：**[**14e4103e78c6dd8bb5b4ff522f0a8a28.gif (950×534)**](https://i-blog.csdnimg.cn/blog_migrate/14e4103e78c6dd8bb5b4ff522f0a8a28.gif)
* **静图：**





1. **核心伪代码**

Quicksort (array, low, high):

if low < high: //递归结束条件

i = low

j = high

key = array[low] //这里的key就是基准（枢轴）

while i < j:

while i < j and array[j] >= key: //从右向左找到第一个小于key的元素

j -= 1

array[i] = array[j] //将小于key的元素放到左边

while i < j and array[i] <= key: // 从左向右找到第一个大于key的元素

i += 1

array[j] = array[i] // 将大于key的元素放到右边

array[i] = key //将基准元素放到最终位置

quicksort(array, low, i - 1) //递归排序基准元素左边的子序列

quicksort(array, i + 1, high) // 递归排序基准元素右边的子序列

1. **测试数据图表：**

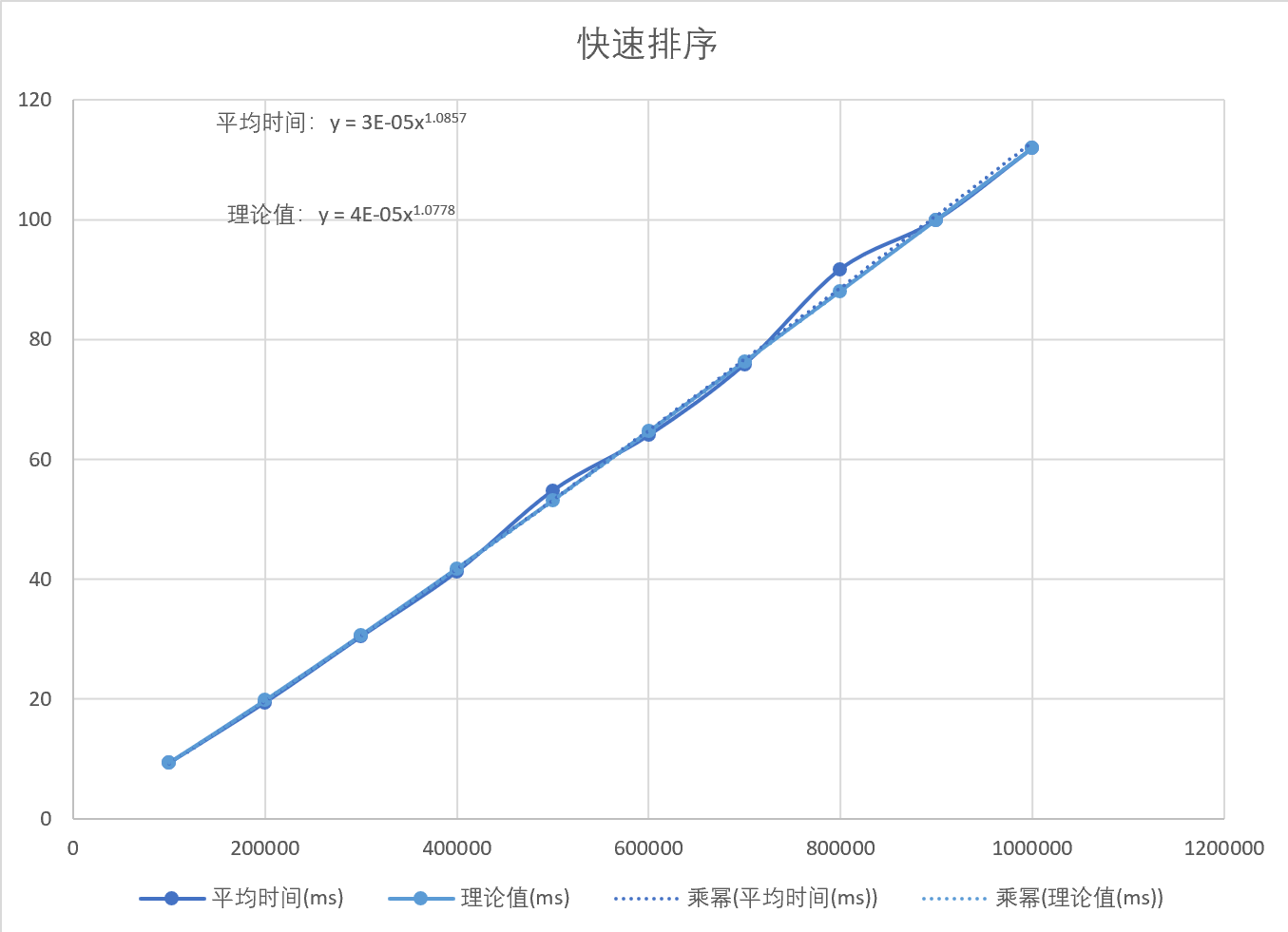
* 理论值计算：

根据快速排序算法的时间复杂度 O(n log n)，并以100万数据规模的理论时间值 T100万=177.557 ms 为基准，公式：Tn=T100万× 。

注意：这里的理论值都是没有优化的算法的理论值



**图像：**



* 分析：

1. 时间复杂度：取决于枢轴（pivot）的选择和数据的分布情况

* 平均：每一层递归的合并操作 O(n) +递归树的深度 log2 n=总时间复杂度为

O(n log n)。

* 最坏：O(n2)

1. 空间复杂度：取决于递归深度

平均：O(log n) 最坏：O(n)

1. 稳定性：

不稳定的，元素在分区过程中会被交换到数组的不同位置，因此相等元素的相对顺序可能会发生改变。

1. 图表分析：

* 理论数据和实测数据分析：

理论值与实际测量值在较大数据规模下较为接近，表明归并排序的时间复杂度模型 O(n log n) 得到了验证。尽管个别数据因为序列本身的随机性而导致偏离理论值，但整体误差都在合理范围内，实际执行时间与理论预测基本吻合。

* 数据规模与平均时间

随着数据规模 n 的增加，平均时间和优化后平均时间都呈现出对数增长的趋势，这符合快速排序的时间复杂度 O(n log n)。

* 性能分析

快速排序在排序大规模的数据集上具有很大优势。快速排序的性能也受到数据分布和枢轴选择策略的影响，比如本次实验测试数据个别数据的偏离，这便是数据分布的影响。

* **五种排序对比分析：**

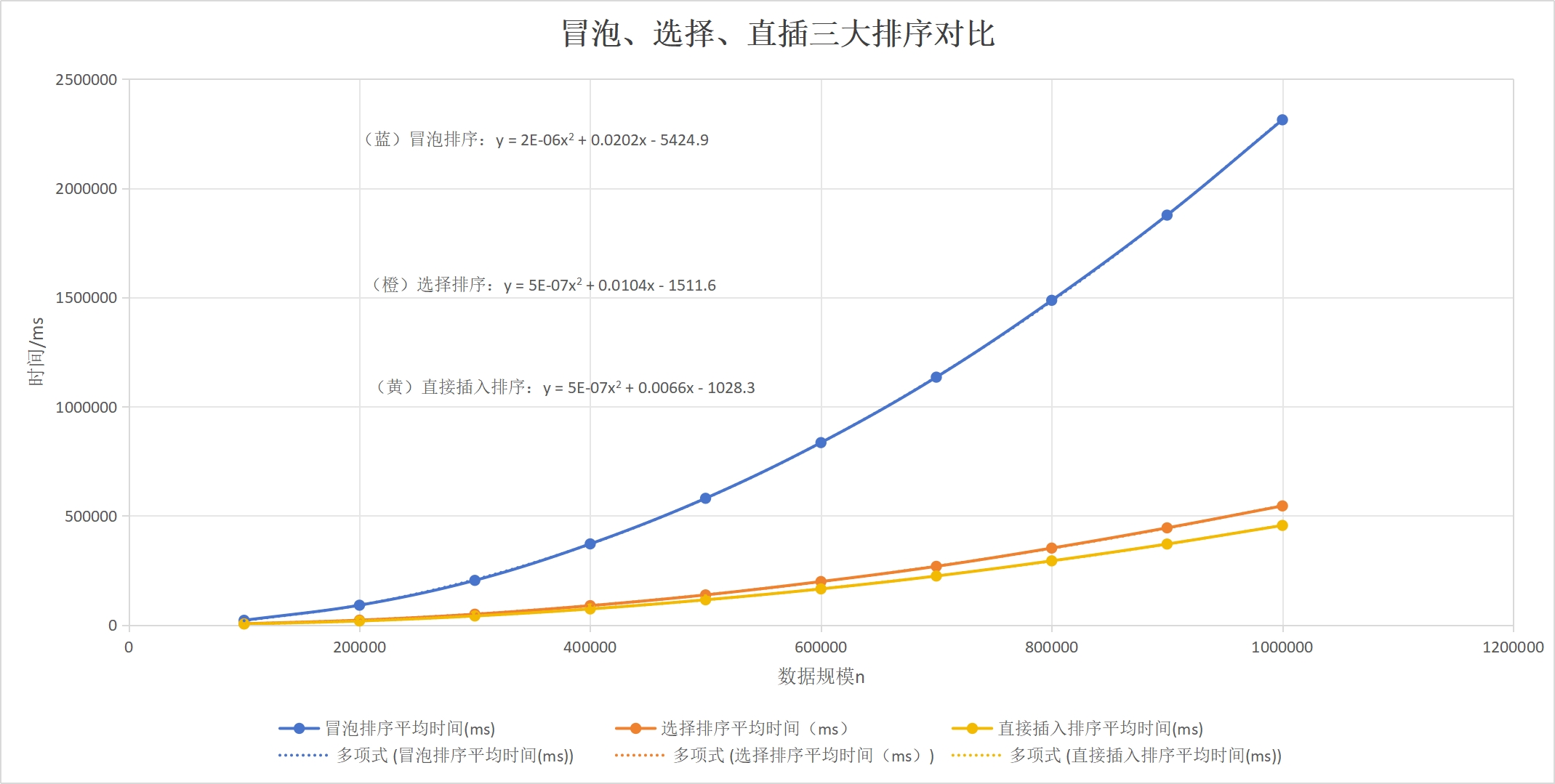
1. **图表：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **数据规模n** | **冒泡排序平均时间(ms)** | **选择排序平均时间（ms）** | **直接插入排序平均时间(ms)** | **希尔排序平均时间（ms）** | **归并排序平均时间（ms）** | **快速排序平均时间(ms)** |
| 100000 | 21444.5 | 5512.18 | 4546.08 | 14.3361 | 14.3361 | 9.35224 |
| 200000 | 90543 | 22003.3 | 18198.7 | 47.6772 | 30.129 | 19.433 |
| 300000 | 204261 | 49325.5 | 41226.9 | 73.2581 | 46.5286 | 30.4261 |
| 400000 | 371375 | 88516.1 | 73405.6 | 105.957 | 64.9764 | 41.2516 |
| 500000 | 580217 | 137755 | 115267 | 131.824 | 81.0861 | 54.7554 |
| 600000 | 835348 | 198907 | 165232 | 163.573 | 98.2606 | 63.9695 |
| 700000 | 1135170 | 268502 | 224312 | 197.427 | 115.716 | 75.8189 |
| 800000 | 1487130 | 352109 | 293733 | 238.253 | 133.441 | 91.6894 |
| 900000 | 1877410 | 444844 | 370699 | 264.208 | 151.217 | 99.872 |
| 1000000 | 2314030 | 545384 | 455965 | 310.462 | 174.892 | 111.849 |

1. **推论：**
2. 运行时间快慢：

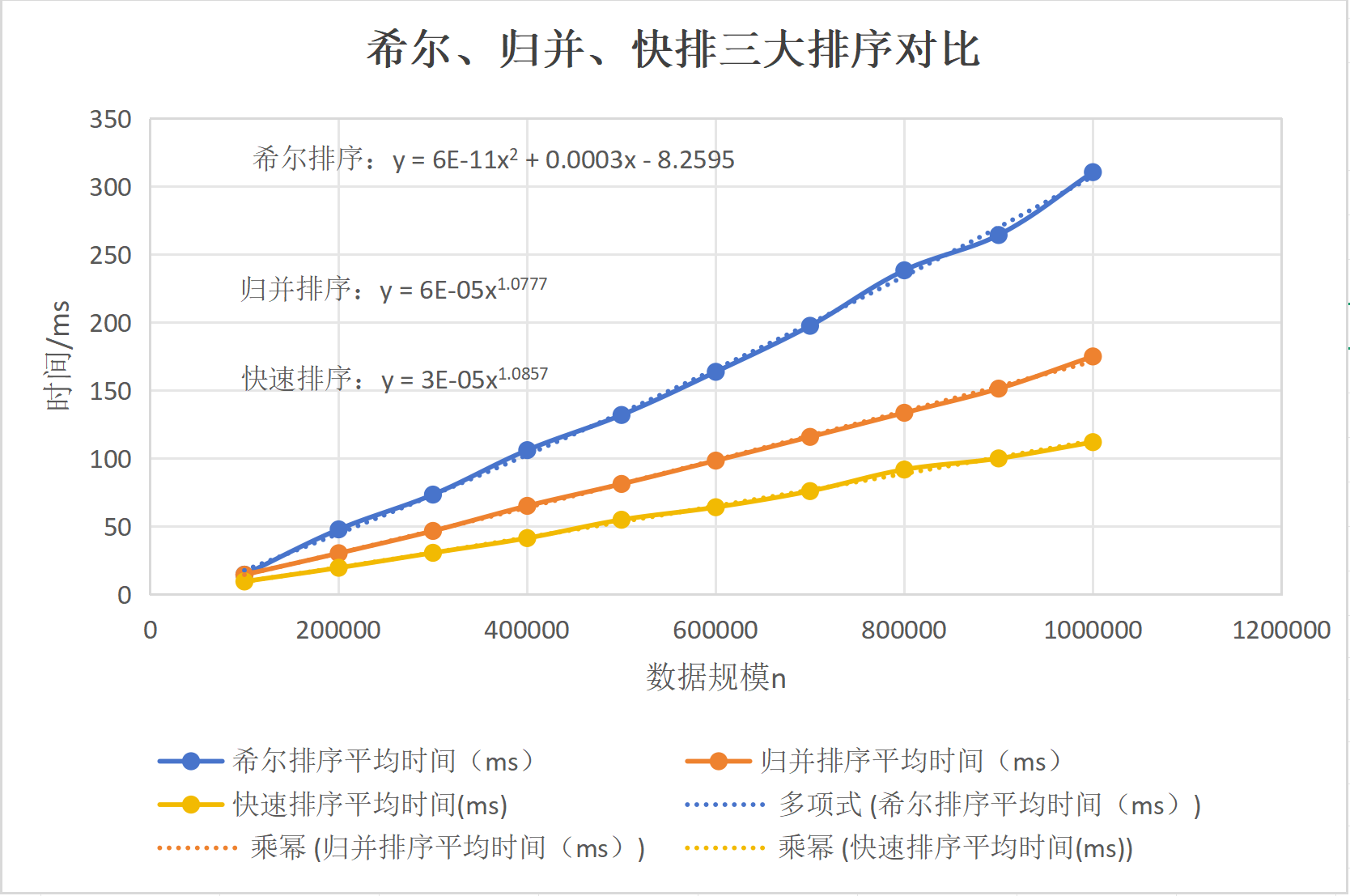
快速排序 > 归并排序 > 插入排序（希尔排序 > 直接插入排序）> 选择排序 > 冒泡排序

1. 冒泡排序在大规模数据排序中效率最低，不适用于处理大规模数据。
2. 快速排序在处理大规模数据时表现最优，推荐作为首选排序算法。
3. 随着数据规模的增大，选择合适的排序算法对提高排序效率至关重要。中等规模时，  
    选择归并排序是追求稳定性和效率的最优选择
4. **冒泡、选择、直插对比：**



分析图像：

1. 增长趋势: 冒泡排序 > 选择排序 > 直接插入排序
2. 我们发现，在较小的数据规模下，冒泡排序的运行时间相对较短，但随着数据规模的增加，其运行时间增长较快。相对应的，选择排序和直接插入排序的增长速度相对冒泡排序要慢得多。
3. 三者对比之下，当数量级很小时（约104左右），三种排序的算法的时间消耗差距不大，但是随着数量级（数据规模）变大，选择排序和直接插入排序明显更具优势。
4. 这三种排序算法在大数据规模下的性能表现都不理想，因此，当数据量较大时，我们应该优先采取快速排序、合并排序、希尔排序等算法。
5. **希尔、归并、快排对比：**



分析图像：

1. 增长趋势: 希尔排序 > 归并排序 > 快速排序
2. 在较小的数据规模下，希尔排序表现较好，运行时间较短，随着数据规模的增加，希尔排序的运行时间增长较快，呈现出二次函数的增长趋势。
3. 在中等和较大的数据规模下，快速排序和归并排序表现更优，运行时间增长较慢，拟合曲线显示其时间复杂度接近 O(n log n)，与他们的理论平均时间复杂度相符
4. 希尔排序和快速排序不稳定，而归并排序是稳定的，因此，在数据规模比较大且要求数据保持稳定性时，优先选择归并排序，而在不考虑稳定性时，优先选择快速排序，而如果有要求空间复杂度较低，可以选择希尔排序（空间复杂度O(1)）。

五种排序对比：

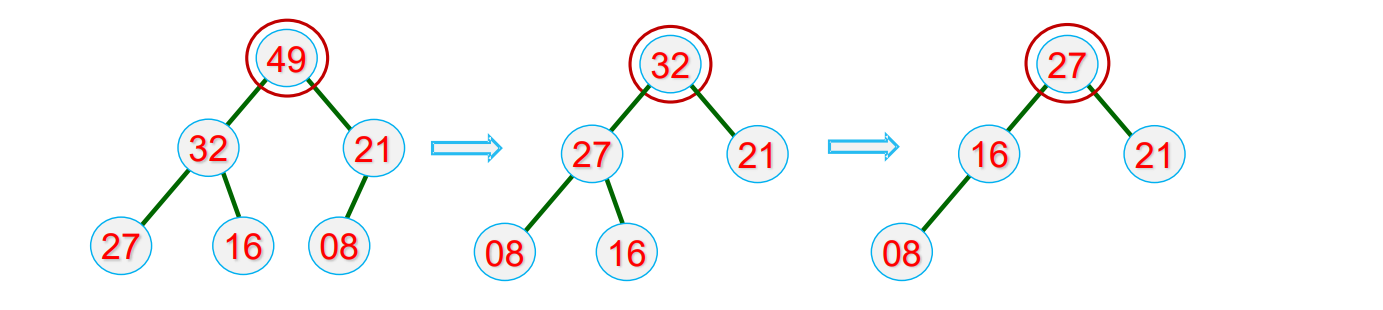
* **Top k 问题**

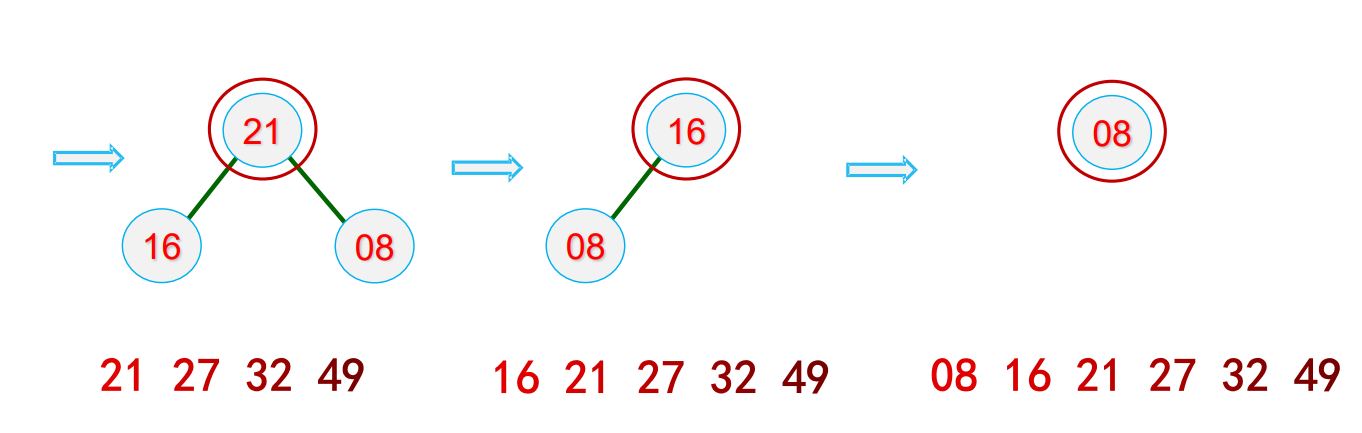
1. **方法一：堆排序**
2. **算法原理描述：**

* 一组待排序的记录，按堆的定义建立初始堆
* 将堆顶记录和第n个记录交换位置
* 堆顶记录被交换后，前n-1个记录如果不再是堆，需将前n-1个待排序记录调整成为一个堆，然后将堆顶记录和第n-1个记录交换位置
* 重复上述操作，直至挑选出k个元素

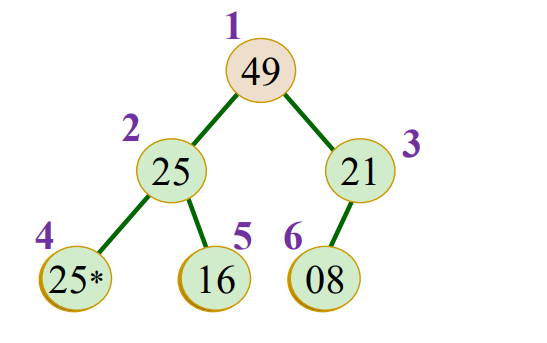
1. **简述：**利用堆顶记录的关键字值最小(或最大)的性质，从当前待排序的记录中依次选取关键字最小(或最大)的记录。
2. **图解：**

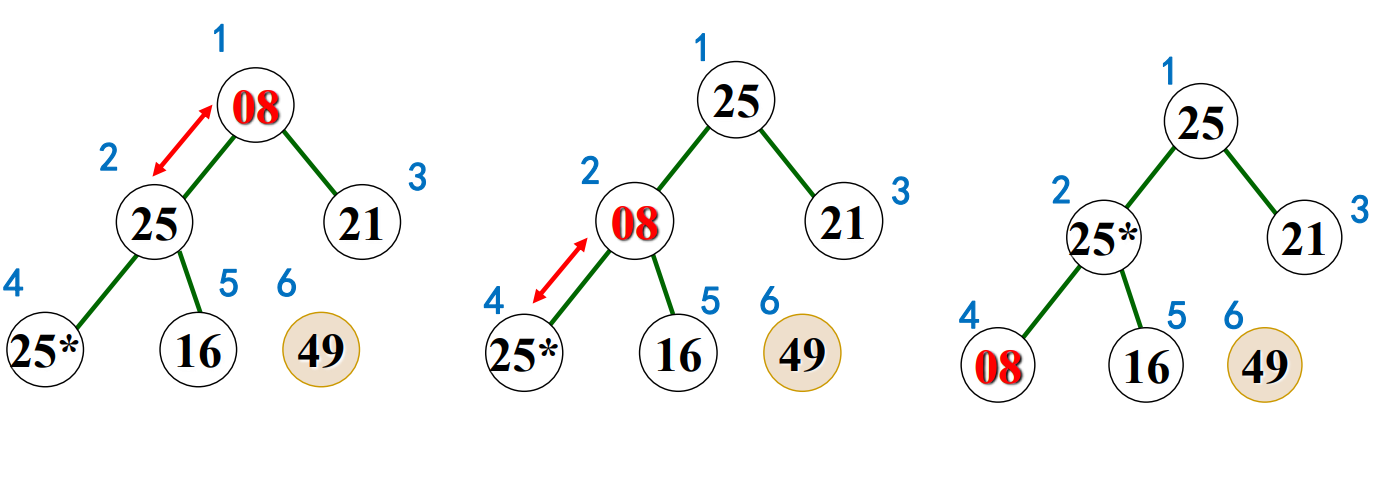
初始序列：49，32，21，27，16，08





筛选过程详解： （初始堆）





1. **核心伪代码：**

* 调整堆：

Min\_heap\_adjust(array, n, i):

j = 2 \* i

while j <= n ：

if j<n && array[j]>array[j+1]:

j += 1

if array[i] < array[j]:

swap(array[i], array[j]) //使得array[j]->两个记录中最小的记录

i = j

j = 2 \* i //结果：堆顶记录为序列的最小记录

* 堆排序过程：

Heap\_sort(array, n, k):

if n <= 1:

return // 空数组或单元素数组不需要排序

for i = n / 2 ->0 (i--) //构建最小堆

Min\_heap\_adjust(array, n, i)

for i=n-1->1(i--) && k>0(k--): // 堆排序

swap(array[0],array[i]) //将堆顶元素（min）与末尾元素交换

Min\_heap\_adjust(array, i, 0) // 调整剩余元素为最小堆

1. **实验结果与分析：**

* 测试数据图表：



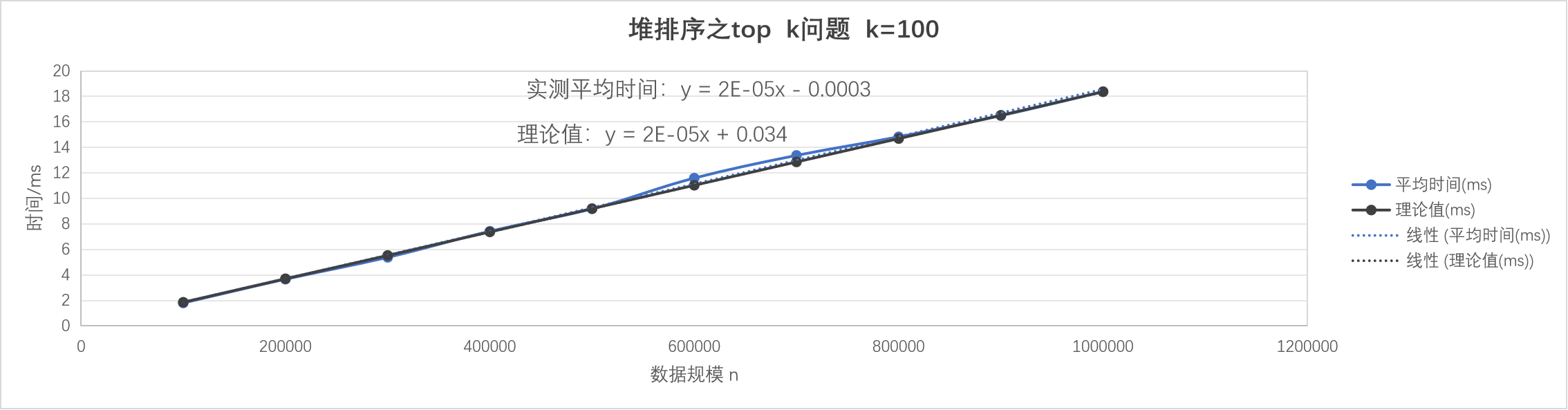
**分析:**

* 对比k=10和 k=100的情况:

在相同的数据规模 n 下，k = 100时的平均时间和理论值通常会比 k = 10时略大

（原因：k 越大，堆调整的次数就越多 T(n,k)=c×(n+k log n)）





**图像分析结论：**

* 结论1：  
  在大多数情况下，平均时间和理论值相近，这表明实验数据与理论预期相符，增长趋势接近线性变化。（k≪n，则 O(n+k∙log n)≈O(n)）
* 结论2：  
  实际平均时间和理论值整体趋势一致且差距不大，只有个别偏差，如 (k=100，n = 60万）时，实际平均时间 (11.5958ms) 大于理论值 (11.0322ms)，这可能是由于在处理较大规模数据和较大 k 值时，堆调整操作更加频繁，受到硬件和软件环境的影响更大，导致实际运行时间有所增加。

**时间复杂度分析：**

* 每个节点调整: O(log n) =》总的调整O(k log n) + 构建堆:O(n)
* 总时间复杂度：

最佳情况（k = O(1)）：O(n)（构建堆主导）。

最坏情况（k = O(n)）：O(n log n)

一般情况：O(n + k log n)

**空间复杂度：**

* O(1) :堆的调整和元素交换等操作——常数级别

**稳定性：**

* 不稳定，在堆调整过程中，相同元素的相对顺序可能会被改变

1. **方法二：快速选择排序**
2. **算法原理描述：**
3. **选择基准值（Pivot）：**

随机从数组中选择一个元素作为基准值（pivot）

1. **划分数组（分区）：**

**重**新排列数组，所有小于等于基准值的元素位于基准值的左边，所有大于等于基准值的元素位于基准值的右边。

1. **确定基准值的位置：**索引 p。
2. **递归或迭代查找：**

p = k-1则找到了目标元素。

p < k-1，则目标元素在右子数组中，在右子数组中继续查找第 k-p-1 小的元素。

p > k-1，则目标元素在左子数组中，在左子数组中继续查找第 k 小的元素。

1. **重复上述步骤：**

不断在子数组中重复选择基准值、划分和查找的过程，直到找到目标元素为止。

1. **概述：**利用分治思想选择一个基准值（pivot）来将数组划分为左右两部分，根据基准值的位置与k的关系，决定是在左子数组还是右子数组中继续查找第k小元素，直到找到目标元素为止。
2. **核心伪代码：**

* 分区函数（Lomuto分区方案）

Partition( array, left, right)

pivot = array[right] // 选择最右元素作为基准

i = left - 1; // 指向小于基准的区域的末尾

for j = left to right

if array[j] <= pivot

i++

swap(array[i],array[j])

swap(array[i+1], array[right])

return i + 1 // 返回基准的最终位置

* 快速选择排序逻辑：

Quickselect( array, left, right, k){

if left == right return left

random = left + rand() % (right - left + 1) // 随机选择基准以避免最坏情况

swap(array[random], array[right]) //交换

pivotIndex = partition(array, left, right)

if pivotIndex == k

return pivotIndex;

else if pivotIndex >k

return Quickselect(array, left, pivotIndex - 1, k)

else

return Quickselect(array, pivotIndex + 1, right, k)

* Top k问题（查找k个最大记录）

topK( array, n, k)

【Array】res

If k<=0 or k>n return {}

targetIndex=Quickselect(array,0,n-1,n-k) //找到第n-k小的位置

For i=targetIndex to n //遍历收集元素

res.push\_back(array[i])

return res

1. **实验结果与分析：**

* 测试数据图表：

时间复杂度：

递推式：T(n)=T(n/2)+O(n) 由主定理法得=≫O(n)

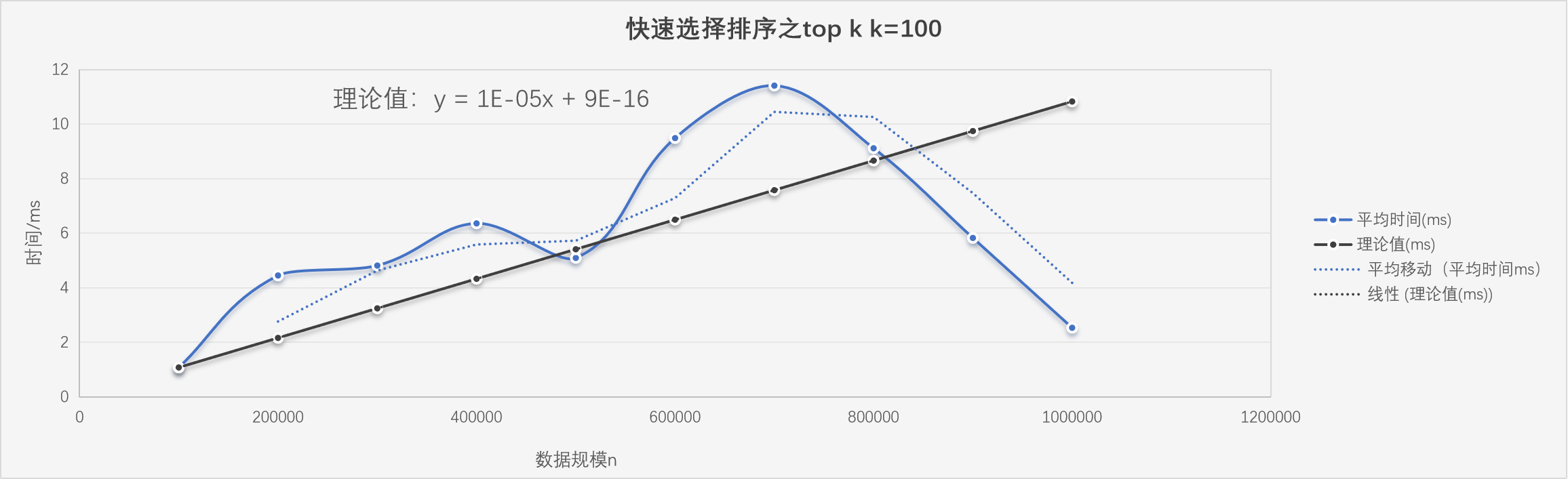
 

测试数据从20组改为500组

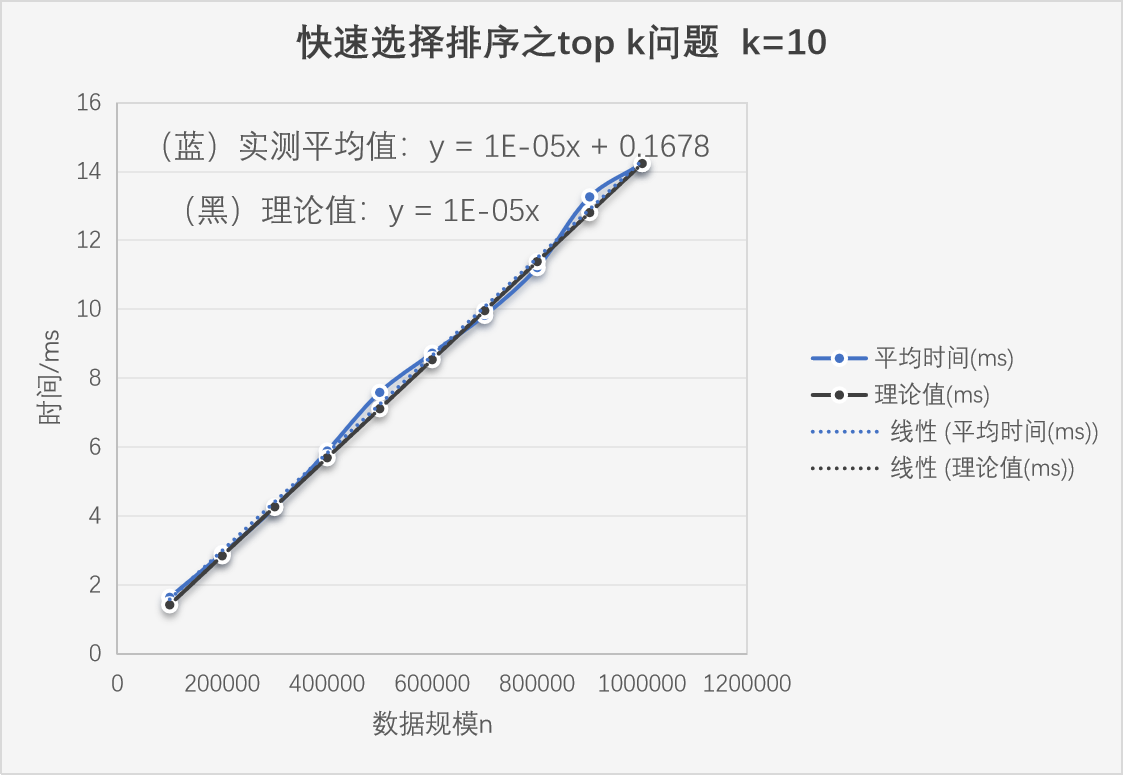
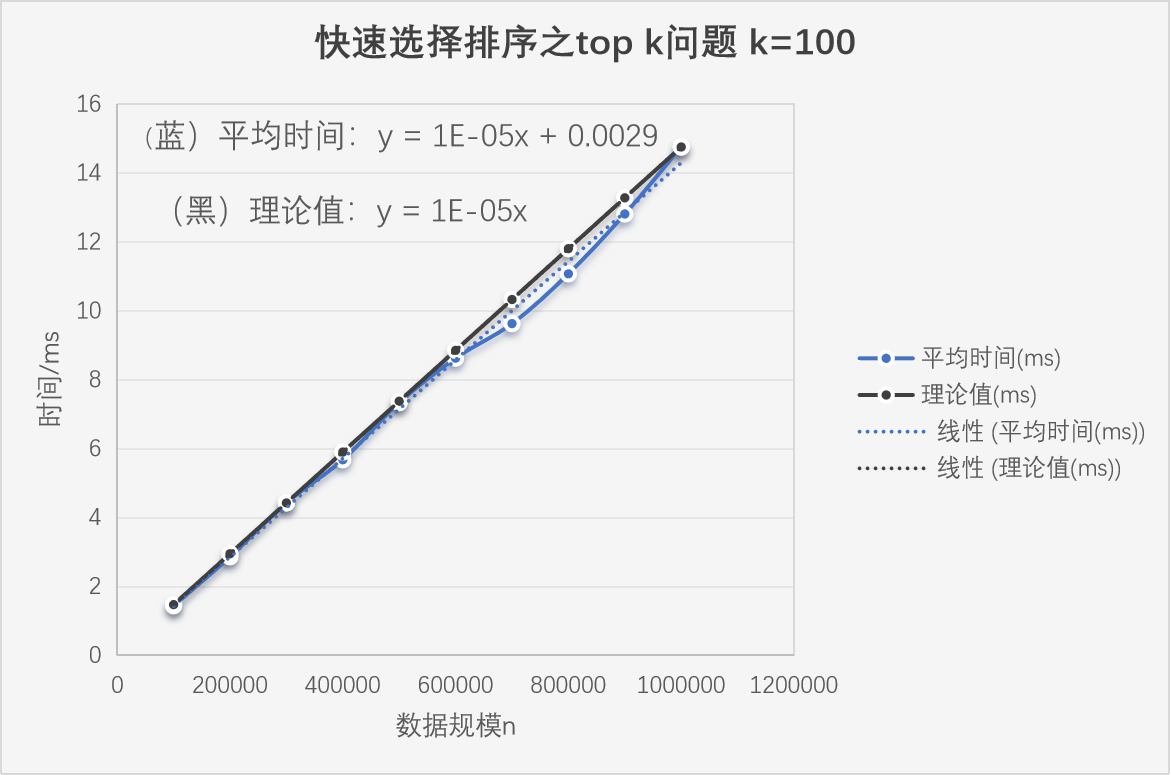
对比：

1. 只选取20组数据取平均值，随机选择pivot对数据误差的产生影响很大--实测数据波动幅度过大
2. 选取500组数据取平均值，随机选择 pivot 的不确定性以及其他偶然因素带来的数据误差确实会被一定程度上平滑，减少误差。

20组数据：（数据波动极大）



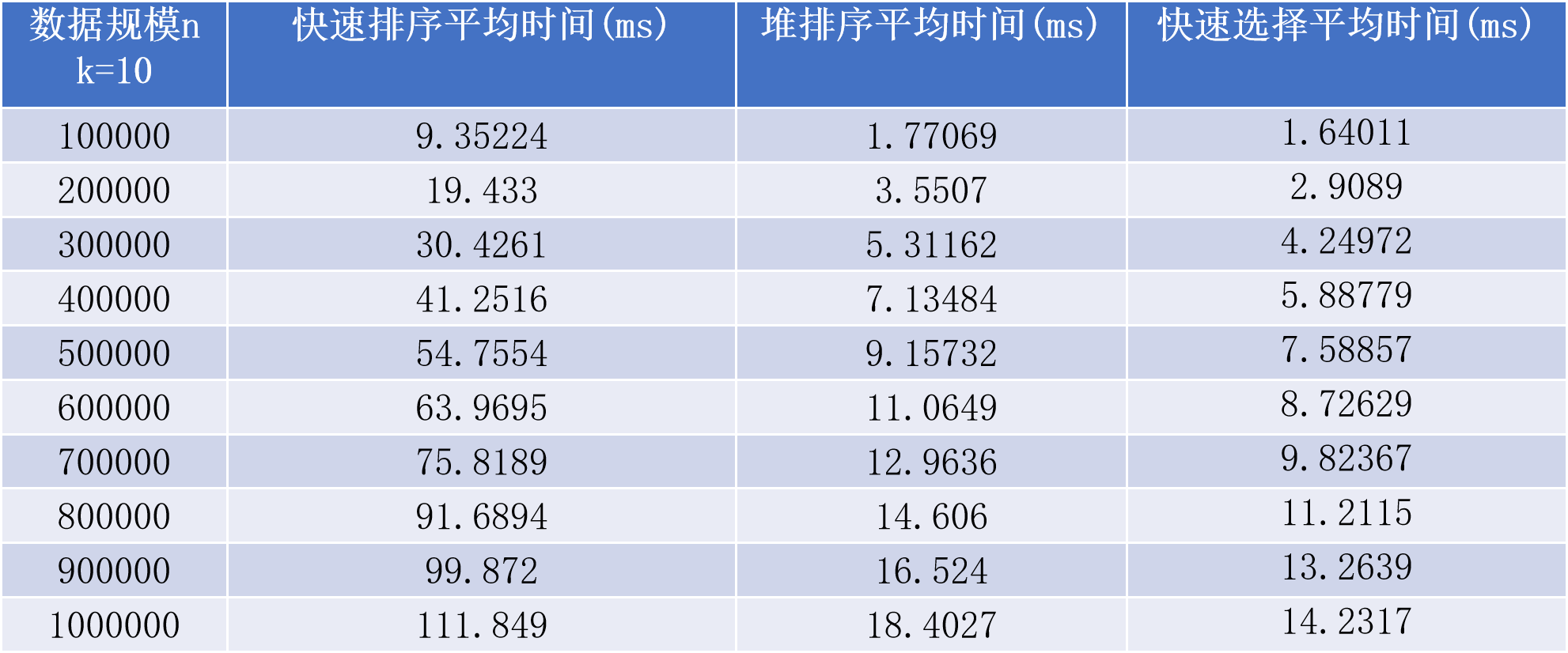
500组数据（数据趋于稳定）

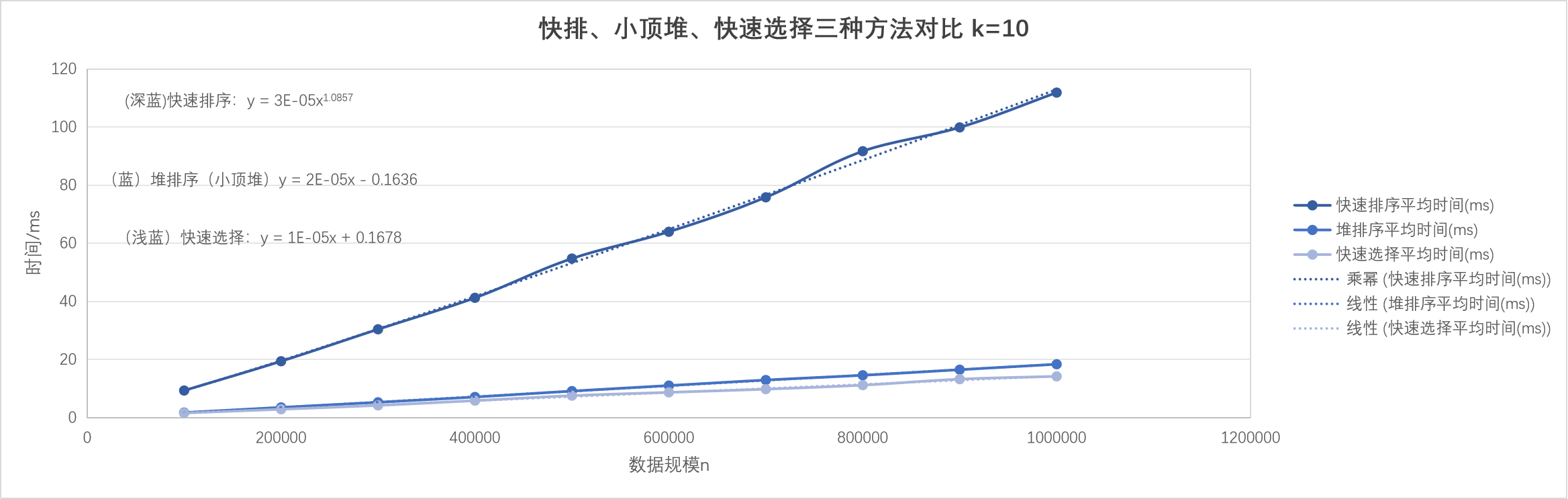
 

分析：

* 结论1：从给定的数据来看，平均时间整体上随着数据规模的增大而呈现出增长的趋势
* 结论2：给定数据中，实测数据波动较小，随机选择基准影响较小，增长趋势接近线性变化，符合其时间复杂度模型O(n)
* 结论3：发现k=10和k=100的拟合曲线一致，说明快速选择的时间复杂度与k无关，为O(n)

1. **三种方法对比分析：**





**分析：**

* 大致速度：  
  快速选择>堆排序(小顶堆)>排序法(快排)
* 平均时间与数据规模的关系：  
  从给定的数据来看，每种方法平均时间整体上随着数据规模的增大而呈现出增长的趋势
* 整体上来说，快速选择排序在解决top k问题上具有较大优势，并且它的平均时间复杂度为O(n),与k无关。而小顶堆为O(n+k log n)，当k比较大时，约为O(n log n)

1. **测试n=10亿，k=10时：**

* 快速排序： 167773.5 ms
* 快速选择排序：1411.57 ms
* 小顶堆：1822.86 ms

**四．实验心得**

* 通过本次实验，我对选择排序、冒泡排序、插入排序、合并排序、快速排序算法原理了解更加深刻，页明白了应该从哪些地方下手优化算法。
* 通过本次实验，我掌握了不同排序算法时间效率的经验分析方法，并且验证了理论分析与经验分析的一致性。
* 排序算法速度：快速排序>合并排序>插入排序（希尔排序>直接插入排序）>选择排序>冒泡排序
* 稳定的排序算法：合并排序、冒泡排序、直接插入排序  
  不稳定的排序算法：快速排序、希尔排序、选择排序
* Top k问题速度：快速选择排序>小顶堆>排序法（快速排序）
* 在实际应用中，不能简单地依据理论时间复杂度来预估算法的实际运行时间，需要考虑到各种实际因素对算法性能的影响。
* 通过本次实验，我学会了如何在分析出理论和实测误差的可能性，并且根据这个可能性去构造一个实验来验证猜测的正确性。

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  年 月 日 |
| 备注： |

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后10日内。