**深 圳 大 学 实 验 报 告**

**课程名称： 算法设计与分析**

**实验名称： 实验1 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（创新班）**

**报告人： XXX 学号： XXXXXXXXXX 班级： 高性能班**

**同组人：**

**指导教师： 杨烜**

**实验时间： 2024.3.10**

**实验报告提交时间： 2024.3.30**

**教务处制**

1. **问题描述**

1、实现以下排序算法：

- 选择排序（Selection Sort）

- 冒泡排序（Bubble Sort）

- 插入排序（Insertion Sort）

- 合并排序（Merge Sort）

- 快速排序（Quick Sort）

对于待排序数组的大小n（n分别取10万、20万、30万、40万、50万）作为输入规模，固定n，随机产生20组测试样本，然后统计上述不同排序算法在这20个样本上的平均运行时间，最终绘制出不同排序算法在这些不同规模的数据上的平均运行时间与输入规模n的关系图。确保横坐标均匀分布。

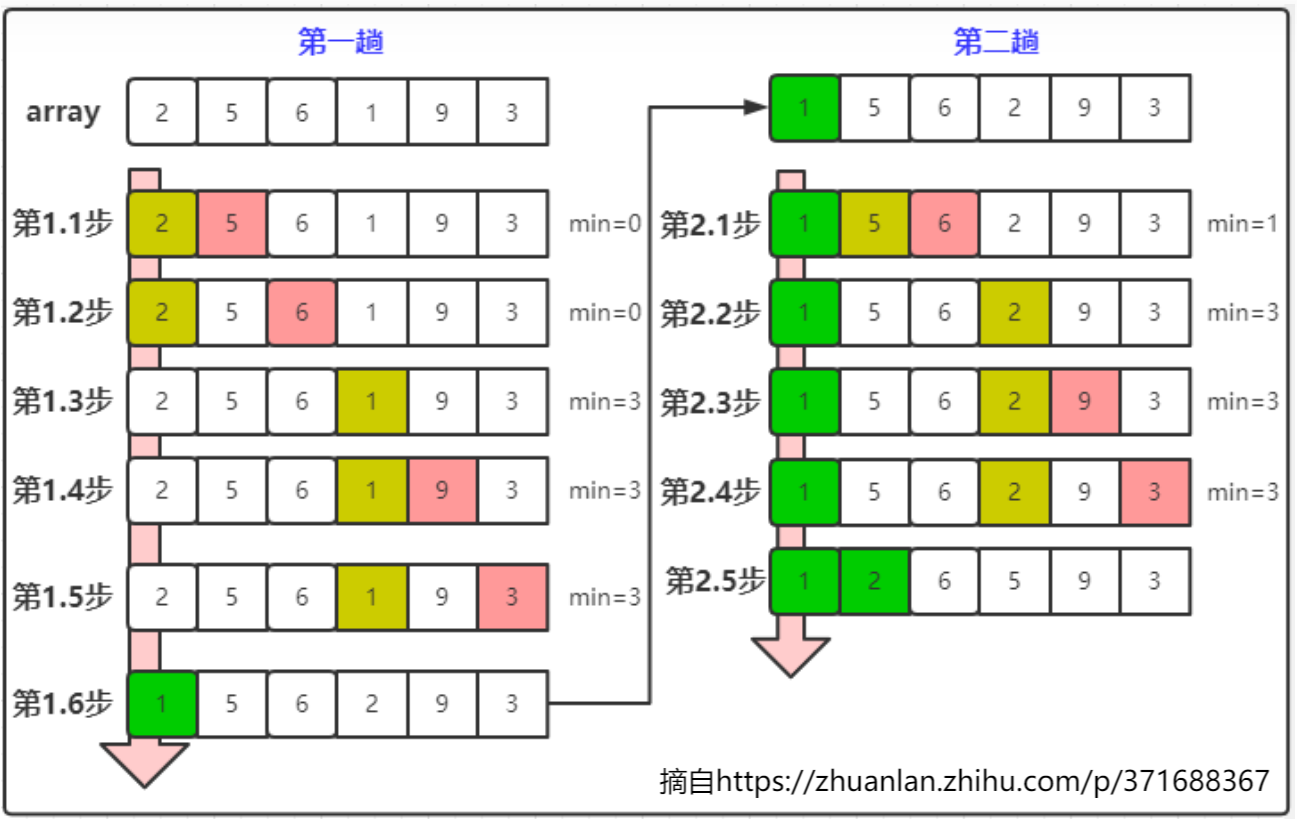
绘制出理论效率分析的曲线和实测的效率曲线。由于实测效率是运行时间，而理论效率是基本操作的执行次数，需要进行对应关系的调整。以输入规模为10万的数据运行时间为基准点，计算输入规模为其他值的理论运行时间，画出不同规模数据的理论运行时间曲线，并与实测的效率曲线进行比较。分析经验分析与理论分析是否一致，如果不一致，请解释存在的原因。

2、 现在有10亿的数据（每个数据四个字节），需要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。

1. **求解问题的算法原理描述**

**问题1：**

1. **选择排序**

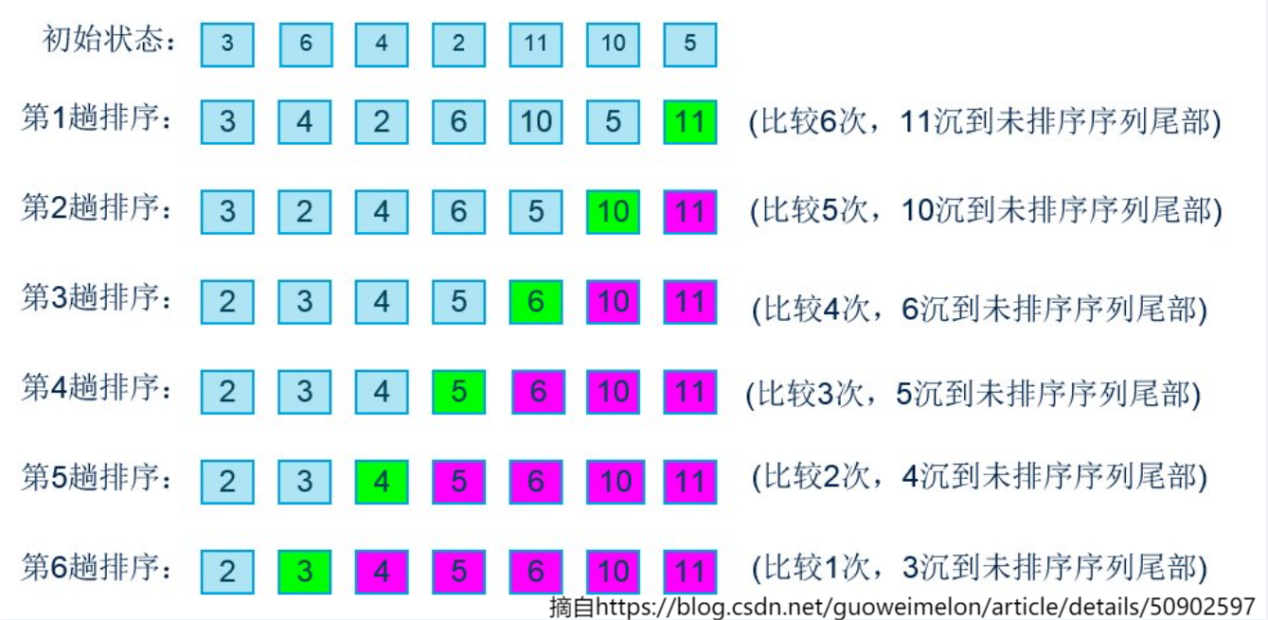


**算法原理：**

遍历要排序的数组，每次找出数组中第i小（或大）的元素，然后将这个元素与数组第i个位置上的元素交换。

**算法的实现细节的解释：**

1. 对于长度为n的数组，选择排序需要进行n-1次遍历来完成排序，其中每次遍历都会在未排序部分确定一个最小元素的位置，并将其放置到已排序部分的末尾。
2. 在升序排序中，假设数组从1开始，在第i次遍历时，会先设第i个元素为最小值min，然后从第i+1个元素开始遍历，比较最小值min和正在遍历元素的大小，若遍历元素小于最小值min，则令min等于正在遍历元素；若遍历元素大于最小值min，则不做操作。然后继续往后遍历，重复上诉操作，直到遍历到第n个元素为止。
3. 选择排序的最坏情况下时间复杂度为O()，这是因为需要进行两层嵌套循环来遍历数组。
4. 选择排序的空间复杂度为O(1)，它只需要固定数量的额外空间来存储临时变量。
5. **冒泡排序**



**算法原理：**

遍历要排序的数组,每次检查相邻两个元素，若前面的元素与后面的元素不符合正确的顺序，就将相邻两个元素交换。当没有相邻的元素需要交换时，即完成排序。

**算法的实现细节的解释：**

1. 对于长度为n的数组，冒泡排序需要进行n-1次遍历来完成排序，每次遍历都会将当前未排序部分的最大（或最小）元素放置到正确的位置。
2. 在第i次遍历中，会从数组第1个元素开始，遍历到第n-i个元素（假设数组从1开始，数组名为arr），每次会比较arr[j]和arr[j+1]，假设要求数组从小到大进行排序，若arr[j]>arr[j+1]，则swap(arr[j],arr[j+1]，且继续比较arr[j+1]和arr[j+2]；若arr[j]<arr[j+1]，则去比较arr[j+1]和arr[j+2],以此类推，直到比较完arr[n-1]和arr[n],才结束该轮循环。
3. 在第1次遍历后，若检查到无任何元素交换，则说明数组已经有序，无需再进行排序，即可提前结束排序。
4. 冒泡排序的时间复杂度为O()，这是因为需要进行两层嵌套循环来遍历数组。
5. 冒泡排序的空间复杂度为O(1)，它只需要固定数量的额外空间来存储临时变量。
6. **插入排序**

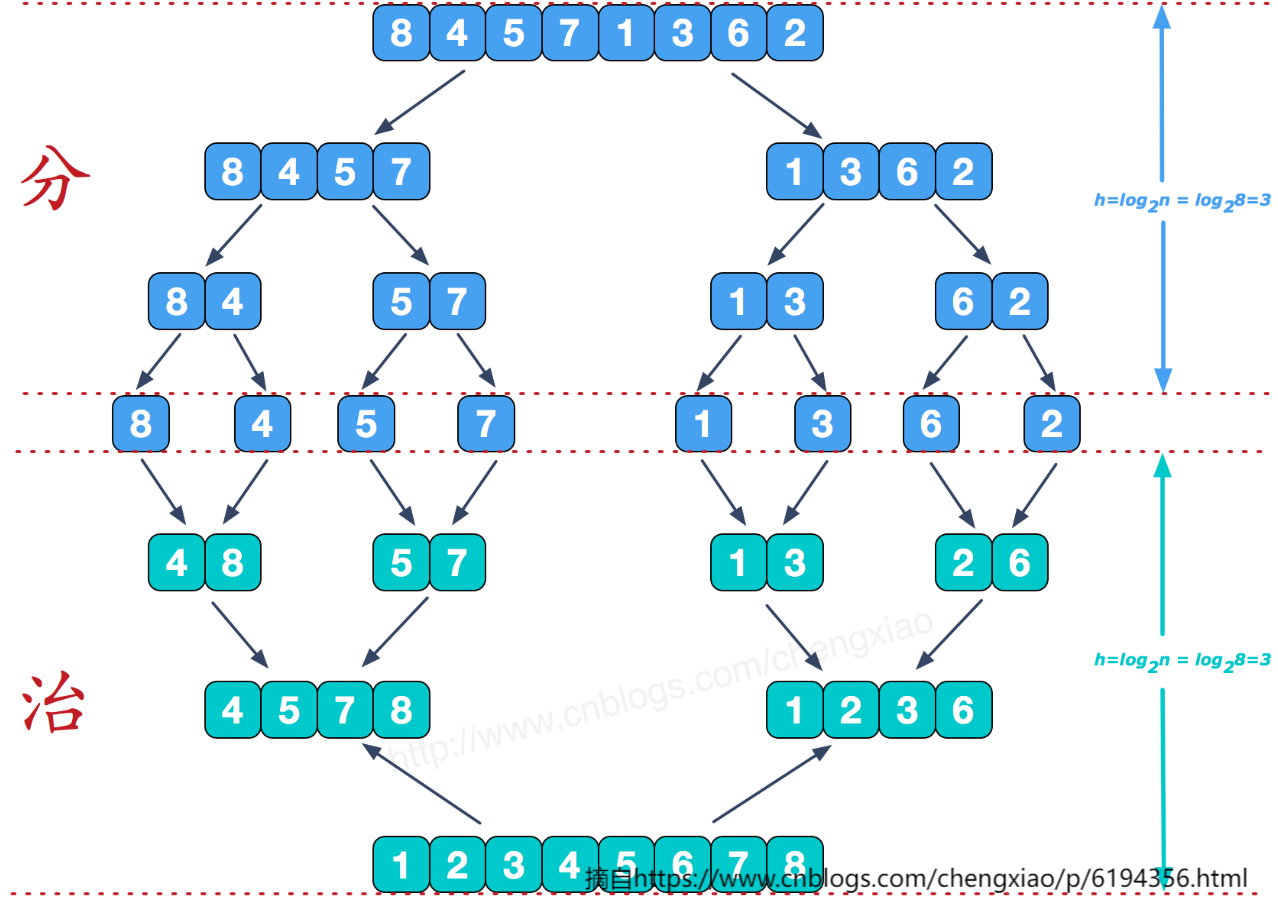


**算法原理：**

遍历要排序的数组,将待排列元素划分为已排序和未排序两部分，每次从未排序的元素中选择一个插入到已排序的元素中的正确位置。

**算法的实现细节的解释：**

1. 对于长度为n的数组，插入排序需要进行n-1次遍历来完成排序，其中每次遍历都会将一个元素插入到已排序部分的正确位置。
2. 在升序排序中，假设数组从1开始，在第i次遍历时，会将第i+1个元素key和前i个已排好序的元素进行比较，从第i个元素开始往前遍历，同时和i+1个元素key进行比较，若遍历到的元素小于第i+1个元素key，则将i+1个元素key插入到该遍历到的元素后面。
3. 所谓的插入，即为在从第i个元素开始往前遍历的过程中，若遍历到的元素arr[j]比第i+1个元素key大，则令arr[j+1]=arr[j]，直到遍历到元素arr[j]比第i+1个元素key小，此时令arr[j+1]=key，则完成插入。
4. 插入排序的时间复杂度为O()，这是因为在最坏情况下，需要进行两层嵌套循环来遍历数组。
5. 插入排序的空间复杂度为O(1)，它只需要固定数量的额外空间来存储临时变量。
6. **合并排序**



**算法原理：**

将待排序的数组分成两个较小的子数组，分别对这两个数组表进行递归排序，然后将已排序的子数组合并成一个有序的数组。这个过程通过递归的方式一直持续下去，直到所有的子数组都变成了只有一个元素的数组，然后通过不断合并有序的子数组，最终得到完全有序的数组。

**算法的实现细节的解释：**

1. 在分解阶段，通过递归将待排序的数组分解成若干个规模较小的子数组，直到每个子数组只包含一个元素。
2. 对子数组进行递归排序（当子数组长度为1时已经有序），即将递归分解得到的子数组进行排序，为后续合并做准备。
3. 对已经排好序的子数组进行合并操作。合并过程中，通过比较两个有序子数组的首元素，选择两个子数组中较小的元素移动到新数组尾部中，直至两个子数组为空，此时，新数组即为合并后的有序数组。这个过程通常需要额外的空间来合并数组。
4. 重复执行合并操作，直到所有子数组都被合并成一个完全有序的数组。
5. 合并排序的时间复杂度为O()，其中n是待排序列表的长度。这是因为在每一层递归中，需要对长度为n的列表进行合并操作，而递归深度为。
6. 合并排序的空间复杂度为O(n)，因为它需要额外的空间来合并数组。
7. **快速排序**



**算法原理：**

通过选取一个基准元素，将数组分割成两部分，左边的部分都小于等于基准元素，右边的部分都大于等于基准元素，然后对这两部分分别递归地进行快速排序。通过不断地分割和排序，最终实现整个数组的排序。

**算法的实现细节的解释：**

1. 从待排序的数组中选取一个元素作为基准元素。通常情况下，选择数组的第一个元素、最后一个元素或者中间元素作为基准元素。
2. 将数组中的元素分割成两部分，一部分是小于等于基准元素的部分，另一部分是大于等于基准元素的部分。这一步可以通过使用两个指针（一前一后）从数组的两端向中间移动，不断交换元素来实现。
3. 对分割后的两部分数组分别递归地进行快速排序。即对小于等于基准元素的部分和大于等于基准元素的部分分别调用快速排序算法。
4. 递归过程中，当子数组的长度小于等于1时，不再进行排序，直接返回。
5. 在平均情况下，快速排序的时间复杂度为O(nlogn)。这是因为每次分割过程都会将列表划分为大致相等大小的两部分，因此递归树的高度大约为n。而每次分割的时间复杂度为O(n)，因此总体时间复杂度为O(nlogn)。
6. 在最坏情况下，快速排序的时间复杂度为O()。这种情况通常发生在数组已经近乎有序的情况下，或者基准元素选择不当（比如始终选择列表的最小或最大元素）。在这种情况下，递归树可能会退化为一棵高度为n的树，每次分割的时间复杂度为O(n)，因此总体时间复杂度为O()。
7. 快速排序的空间复杂度为O(1)，它只需要固定数量的额外空间来存储临时变量。

**问题2：**

1. **最大堆算法**

**算法原理：**

通过构建一个二叉堆（通常是最大堆），然后重复执行以下步骤：将堆顶元素（最大元素）与堆中的最后一个元素交换位置，然后将堆的大小减小，并将交换后的堆顶元素重新调整为最大堆。重复这个过程，直到堆的大小为len - k，即最大的k个数已被挑选出来。

**算法的实现细节的解释：**

1. 首先将待排序数组构建成一个最大堆。这可以通过从数组的中间位置开始向前遍历，对每个非叶子节点进行堆调整（调整为最大堆）来实现。堆调整的过程是先比较左右子节点的大小，挑选出较大的子节点后，再让较大的子节点与父亲节点比较，若较大的子节点的大小小于父亲节点的大小，则退出非叶子节点的堆调整；否则，交换较大的子节点与父亲节点，然后递归地对交换位置后的子节点进行堆调整，直到满足最大堆的性质。
2. 构建好最大堆后，将堆顶元素（最大值）与堆中最后一个元素交换位置，然后将堆的大小减1，即排除掉最大值。接着再对堆顶元素进行堆调整，将其调整为最大堆。重复这个过程，直到排除掉k个最大值，数组arr[len-k]到arr[len-1]即为问题所求的最大的k个数。
3. 堆排序的建立堆最坏的时间复杂度为O(n)，在堆调整中，由于只需要进行k-1次堆调整就可以得到最大的k个数，假设每次堆调整都是最坏的情况，即根节点需要从堆顶调整到堆底，且不考虑其他操作的影响，则k-1次堆调整的时间复杂度的计算如下

由于n的取值到1000000000，则

总的时间复杂度为。

1. 堆排序的空间复杂度为O(n)，它需要固定数量的额外空间来存储10亿个数，空间开销大。
2. **基于选择排序的算法**

**算法原理：**

遍历数组，每次找出数组中第i大的元素，然后将这个元素与数组第i个位置上的元素交换。循环遍历k次后，数组arr[1]到arr[k]（假设数组下标从1开始）为从大到小排序的最大的k个数。

**算法的实现细节的解释：**

1. 选择排序每次遍历都会在未排序部分确定一个最大元素的位置，并将其放置到已排序部分的末尾，所以总共只需要循环遍历k次，数组中最大的k个数即可确定。
2. 假设数组从1开始，在第i次遍历时，会先设第i个元素为最小值max，然后从第i+1个元素开始遍历，比较最大值max和正在遍历元素的大小，若遍历元素大于最大值max，则令max等于正在遍历元素；若遍历元素小于最大值max，则不做操作。然后继续往后遍历，重复上诉操作，直到遍历到第n个元素为止。
3. 由于只遍历k次数组，由于第i次遍历序遍历i个元素，在不考虑其他操作的影响下，该算法的时间复杂度的计算如下
4. 选择排序的空间复杂度为O(n)，它需要固定数量的额外空间来存储10亿个数，空间开销大。
5. **基于快速排序的算法**

**算法原理：**

通过选取一个基准元素，将数组分割成两部分，左边的部分都小于等于基准元素，右边的部分都大于等于基准元素，然后优先对右边部分进行递归的快速排序，当完成了数组下标n-k+1到n（假设数组下标从1开始）的递归排序后，此时则已经挑选出数组中最大的k个元素。

**算法的实现细节的解释：**

1. 从待排序的数组中选取一个元素作为基准元素。通常情况下，选择数组的第一个元素、最后一个元素或者中间元素作为基准元素。
2. 将数组中的元素分割成两部分，一部分是小于等于基准元素的部分，另一部分是大于等于基准元素的部分。这一步可以通过使用两个指针（一前一后）从数组的两端向中间移动，不断交换元素来实现。
3. 对分割后的两部分数组分别递归地进行快速排序，但优先对右边部分进行递归的快速排序，这是因为问题要求挑选出最大的十个元素，此时右边部分均比左边部分要大，故先递归右边。
4. 在递归的过程中，若发现基准元素的下标小于等于n-k+1，high==n，且已经完成了[基准元素下标+1，high]的递归排序工作，说明数组中最大的k个元素已经被挑选出来，则可令标记flag=1。
5. 为了让快速排序在找到数组中最大的k个元素后能尽早结束工作，故设定标记flag，在递归过程中，若检测到flag=1,则说明已经找到数组中最大的k个元素，直接返回即可。
6. 递归过程中，当子数组的长度小于等于1时，不再进行排序，直接返回。
7. 在最优的情况下，数组在第一次确定基准元素时，基准元素的下标为arr[n-k+1]，则只需要再完成下标n-k+2到n的递归排序，就能挑选出数组中最大的十个元素。假设每次递归排序的复杂度为O(len)，其中len为递归排序前子数组的长度，不考虑其他操作对复杂度的影响，则最优情况下的时间复杂度为O(n+k-1)=O(n)。
8. 在每次分割子数组均平均的情况下，即基准元素把子数组对半划分。假设每次递归排序的复杂度为O(len)，其中len为递归排序前子数组的长度，由于优先对右边部分进行递归排序，且在找到数组中最大的k个元素就退出函数，故到目标区间段前递归每层的时间复杂度为len；当递归到目标区间段（基准元素的下标小于等于n-k+1，high==n，即k），由于该区间段需要继续递归到长度为1的区间段，递归的深度为，每层的操作数为，故目标区间段的操作总数为，不考虑其他操作对复杂度的影响，该情况下的时间复杂度的计算如下

故在数据足够随机的情况下，快速排序的时间复杂度为。

1. 快速排序的空间复杂度为O(n)，它需要固定数量的额外空间来存储10亿个数，空间开销大。
2. **最小堆算法**

**算法原理：**

取10亿个数中前k个数在内维护一个长度为k的最小堆，然后从剩下的数中挨个读取数字并和堆顶比较，如果比堆顶小则直接丢弃，否则替换堆顶后调整最小堆。遍历完文件中所有的数字后，最小堆中的k个数就是所求的最大的k个数。

**算法的实现细节的解释：**

1. 首先，从10亿个数中读取前k个数，构建一个大小为k的最小堆。这可以通过将前k个数放进一个长度为k的数组中，从数组的中间位置开始向前遍历，对每个非叶子节点进行堆调整（调整为最大堆）来实现。堆调整的过程是先比较左右子节点的大小，挑选出较小的子节点后，再让较小的子节点与父亲节点比较，若较小的子节点的大小大于父亲节点的大小，则退出非叶子节点的堆调整；否则，交换较小的子节点与父亲节点，然后递归地对交换位置后的子节点进行堆调整，直到满足最小堆的性质。
2. 接下来，读取剩余的数。对于每个读取到的数，与堆顶的数比较：如果当前数小于等于堆顶的数，则丢弃该数，因为它不可能是前k大的数；如果当前数大于堆顶的数，则将堆顶的数替换为当前数，并进行最小堆的调整操作，以保持堆的性质。
3. 当所有数字都被处理完毕后，堆中剩下的k个数即为所求的前k大的数。
4. 构建初始最小堆的时间复杂度为O(k)，遍历剩余数的时间复杂度为O((n-k)logk)，整体时间复杂度为 。
5. 该算法的空间复杂度为O(k)，它只需要固定数量的额外空间来存储最小堆即可，这是本算法最大的优点之一，在现实中，面对海量的数据，计算机可能无法一下子存下规模过大的数据，但通过建立最小堆后，无需一下子接受完全部数据，只需要数据一个一个传过来进行比较即可，节约了大量的存储空间。
6. **算法实现的核心伪代码**

**问题1：**

1. **选择排序**

**改进前**

|  |
| --- |
| *for i from 0 to n - 2 do*  *min : = i*  *for j from i + 1 to n - 1 do*  *if arr[j] < arr[min] then*  *min : = j*  *swap(arr[i], arr[min])* |

**改进后**

|  |
| --- |
| *for i from 0 to n - 2 do*  *min : = i*  *for j from i + 1 to n - 1 do*  *if arr[j] < arr[min] then*  *min : = j*  *tmp : = arr[i]*  *arr[i] : = arr[min]*  *arr[min] : = tmp* |

1. **冒泡排序**

**改进前**

|  |
| --- |
| *for i from 0 to n - 2 do*  *flag := 1*  *if i = 1 and flag = 1 then*  *break*  *for j from 0 to n - 2 - i do*  *if arr[j] > arr[j + 1] then*  *swap(arr[j], arr[j + 1])*  *flag := 0* |

**改进后**

|  |
| --- |
| *for i from 0 to n - 2 do*  *flag := 1*  *if i = 1 and flag = 1 then*  *break*  *for j from 0 to n - 2 - i do*  *if arr[j] > arr[j + 1] then*  *tmp : = arr[j]*  *arr[j] : = arr[j + 1]*  *arr[j + 1] : = tmp*  *flag := 0* |

1. **插入排序**

|  |
| --- |
| *for i from 1 to n - 1 do*  *key : = arr[i]*  *j : = i - 1*  *while j >= 0 and key < arr[j] do*  *arr[j + 1] : = arr[j]*  *j : = j - 1*  *arr[j + 1] : = key* |

1. **合并排序**

**改进前**

|  |
| --- |
| *// 合并操作*  *procedure Merge(arr: array of integers, front : integer, mid : integer, end : integer)*  *tmp\_arr1 : = new integer[mid - front + 1]*  *tmp\_arr2 : = new integer[end - mid]*  *for i from 0 to mid - front do*  *tmp\_arr1[i] : = arr[front + i]*  *for i from 0 to end - mid - 1 do*  *tmp\_arr2[i] : = arr[mid + 1 + i]*  *idxLeft : = 0*  *idxRight : = 0*  *i : = front*  *while idxLeft < mid - front + 1 and idxRight < end - mid do*  *if tmp\_arr1[idxLeft] < tmp\_arr2[idxRight] then*  *arr[i] : = tmp\_arr1[idxLeft]*  *idxLeft : = idxLeft + 1*  *else*  *arr[i] : = tmp\_arr2[idxRight]*  *idxRight : = idxRight + 1*  *i : = i + 1*  *while idxLeft < mid - front + 1 do*  *arr[i] : = tmp\_arr1[idxLeft]*  *idxLeft : = idxLeft + 1*  *i : = i + 1*  *while idxRight < end - mid do*  *arr[i] : = tmp\_arr2[idxRight]*  *idxRight : = idxRight + 1*  *i : = i + 1*  *delete tmp\_arr1*  *delete tmp\_arr2*  *// 合并排序*  *procedure Merge\_Sort(arr: array of integers, front : integer, end : integer)*  *if front >= end then*  *return*  *mid : = (front + end) / 2*  *Merge\_Sort(arr, front, mid)*  *Merge\_Sort(arr, mid + 1, end)*  *Merge(arr, front, mid, end)* |

**改进后**

|  |
| --- |
| *// 合并操作*  *Procedure Merge(arr: array of int, tmp : array of int, front : integer, mid : integer, end : integer)*  *Declare idxLeft, idxRight, i as integer*  *idxLeft : = 0*  *idxRight : = 0*  *For i : = front to end do*  *if idxLeft = mid - front + 1 or idxRight = end - mid then*  *Break*  *if arr[idxLeft] < arr[idxRight] then*  *tmp[i] : = arr[idxLeft]*  *idxLeft : = idxLeft + 1*  *else*  *tmp[i] : = arr[idxRight]*  *idxRight : = idxRight + 1*  *while idxLeft < mid - front + 1 do*  *tmp[i] : = arr[idxLeft]*  *i : = i + 1*  *idxLeft : = idxLeft + 1*  *while idxRight < end - mid do*  *tmp[i] : = arr[idxRight]*  *i : = i + 1*  *idxRight : = idxRight + 1*  *for i : = front to end do*  *arr[i] : = tmp[i]*  *// 合并排序*  *Procedure Merge\_Sort(arr : array of int, t : array of int, left : integer, right : integer)*  *if left < right then*  *mid : = (left + right) / 2*  *Merge\_Sort(arr, t, left, mid)*  *Merge\_Sort(arr, t, mid + 1, right)*  *Merge(arr, t, left, mid, right)* |

1. **快速排序**

|  |
| --- |
| *// 返回基准元素并对子数组进行相对大小排序*  *function Partition1(arr: array of integers, low : integer, high : integer)->integer:*  *pivot: = arr[low]*  *while low < high do*  *while low < high and arr[high] >= pivot do*  *high : = high - 1*  *arr[low] : = arr[high]*  *while low < high and arr[low] <= pivot do*  *low : = low + 1*  *arr[high] : = arr[low]*  *arr[low] : = pivot*  *return low*  *// 快速排序*  *procedure Quick\_Sort(arr : array of integers,*  *low : integer, high : integer)*  *if low < high then*  *pivot : = Partition1(arr, low, high)*  *Quick\_Sort(arr, low, pivot - 1)*  *Quick\_Sort(arr, pivot + 1, high)* |

**问题2：**

1. **最大堆算法**

|  |
| --- |
| *// 堆调整*  *procedure Max\_Heapify(arr: array of int, start : integer, end : integer)*  *dad : = start*  *son : = dad \* 2 + 1*  *while son ≤ end do*  *if son + 1 ≤ end and arr[son] < arr[son + 1] then*  *son : = son + 1*  *if arr[dad] > arr[son] then*  *return*  *else*  *stmp : = arr[dad]*  *arr[dad] : = arr[son]*  *arr[son] : = tmp*  *dad : = son*  *son : = dad \* 2 + 1*  *// 最大堆算法，arr[len-10]到arr[len-1]即为最大的十个数*  *procedure Heap\_Sort(arr : array of int, len : integer)*  *for i from len / 2 - 1 downto 0 do*  *Max\_Heapify(arr, i, len - 1)*  *tmp : = arr[0]*  *arr[0] : = arr[len - 1]*  *arr[len - 1] : = tmp*  *for i from len - 2 downto len - 11 do*  *Max\_Heapify(arr, 0, i)*  *tmp : = arr[0]*  *arr[0] : = arr[i]*  *arr[i] : = tmp* |

1. **基于选择排序的算法**

|  |
| --- |
| *// 基于选择排序的算法，arr[0]到arr[9]即为最大的十个数*  *for i from 0 to 9 do*  *maxIndex := i*  *for j from i + 1 to n - 1 do*  *if arr[j] > arr[maxIndex]then*  *maxIndex := j*  *tmp : = arr[i]*  *arr[i] : = arr[maxIndex]*  *arr[maxIndex] : = tmp* |

1. **基于快速排序的算法**

|  |
| --- |
| *flag : = 0 // 标记，若flag==1，说明快速排序已选出数组中最大的十个数，可以*  *// 退出函数*  *// 返回基准元素并对子数组进行相对大小排序*  *function Paritition1(arr : array of int, low : integer, high : integer)->integer*  *pivot : = arr[low]*  *while low < high do*  *while low < high and arr[high] ≥ pivot do*  *high : = high - 1*  *arr[low] : = arr[high]*  *while low < high and arr[low] ≤ pivot do*  *low : = low + 1*  *arr[high] : = arr[low]*    *arr[low] : = pivot*  *return low*  *// 基于快速排序的算法，arr[n-10]到arr[n-1]即为最大的十个数*  *procedure Quick\_Sort(arr : array of int, low : integer, high : integer, n : integer)*  *if flag = 1 then*  *return*  *if low < high then*  *pivot : = Paritition1(arr, low, high)*  *Quick\_Sort(arr, pivot + 1, high, n)*  *if flag = 1 then*  *return*    *if pivot <= n - 10 and high = n - 1 then*  *flag : = 1*  *return*  *Quick\_Sort(arr, low, pivot - 1, n)*  *if flag = 1 then*  *return* |

1. **最小堆算法**

|  |
| --- |
| *// 堆调整*  *Procedure Min\_Heapify(arr: array of int, start : integer, end : integer)*  *dad : = start*  *son : = dad \* 2 + 1*  *while son ≤ end do*  *if son + 1 ≤ end and arr[son] > arr[son + 1] then*  *son : = son + 1*  *if arr[dad] < arr[son] then*  *return*  *else*  *tmp : = arr[son]*  *arr[son] : = arr[dad]*  *arr[dad] : = tmp*  *dad : = son*  *son : = dad \* 2 + 1*  *// 最小堆算法*  *Procedure Heap\_Sort(arr : array of int, k : integer, len : integer)*  *for i from k / 2 - 1 downto 0 do*  *Min\_Heapify(arr, i, k - 1)*  *for i from k to len - 1 do*  *if arr[i] > arr[0] then*  *arr[0] : = arr[i]*  *Min\_Heapify(arr, 0, k)* |

1. **算法测试结果及效率分析**

**问题1：**

1. **选择排序**

根据20组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 5.523700 | 5.523700 |
| 200000 | 21.787800 | 22.094800 |
| 300000 | 48.960450 | 49.713300 |
| 400000 | 88.318000 | 88.379200 |
| 500000 | 141.632100 | 138.092500 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 5.523700（s）

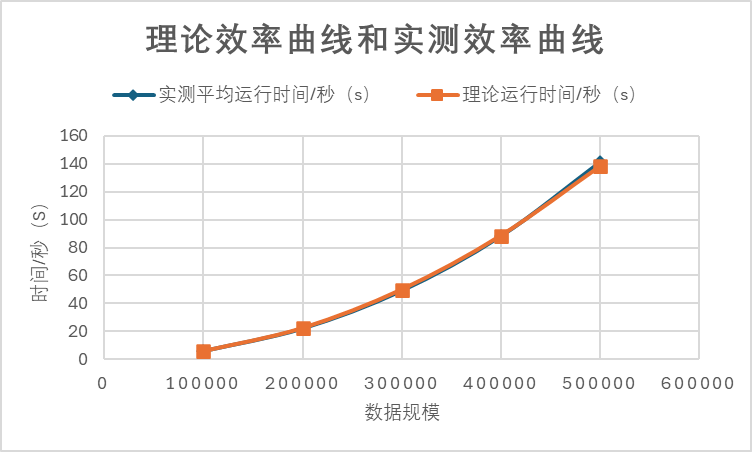
n = 200000时，理论运行时间 = 5.523700（s）

n = 300000时，理论运行时间 = 49.713300（s）

n = 400000时，理论运行时间 = 88.379200（s）

n = 500000时，理论运行时间 = 138.092500（s）

理论效率曲线和实测效率曲线



通过观察理论效率曲线和实测效率曲线，不难发现，理论值与实际值误差较小，且图像符合O()的二次曲线。考虑到代码中使用了swap函数，可能会增加算法运行时间上的开销，故进行改进。

将swap函数去除，直接手写交换代码，得到的新的结果如下

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 5.674000 | 5.674000 |
| 200000 | 22.538000 | 22.696000 |
| 300000 | 50.824000 | 51.066000 |
| 400000 | 89.909500 | 90.784000 |
| 500000 | 140.182500 | 141.850000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 5.674000（s）

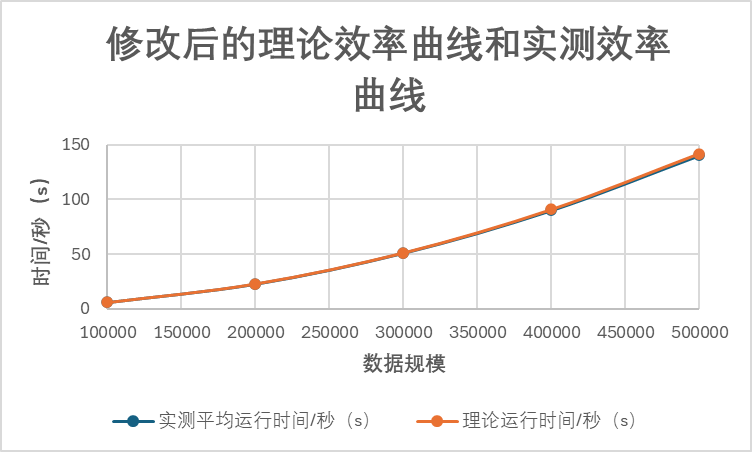
n = 200000时，理论运行时间 = 22.696000（s）

n = 300000时，理论运行时间 = 51.066000（s）

n = 400000时，理论运行时间 = 90.784000（s）

n = 500000时，理论运行时间 = 141.850000（s）

修改后的理论效率曲线和实测效率曲线



根据上图，发现修改后与修改前的结果并无太大差别，这应该是由于选择排序中swap函数调用次数较少，对算法运行时间的开销并未产生明显的影响。

1. **冒泡排序**

根据20组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 22.767800 | 22.767800 |
| 200000 | 96.211000 | 91.071200 |
| 300000 | 215.092950 | 204.910200 |
| 400000 | 383.856200 | 364.284800 |
| 500000 | 603.968150 | 569.195000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 22.767800（s）

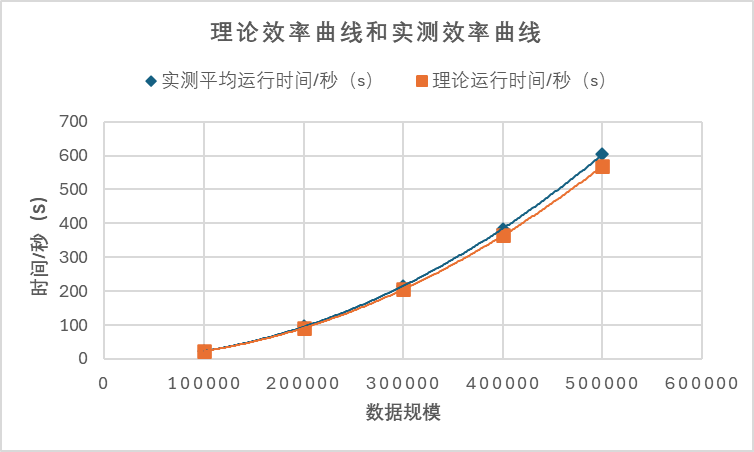
n = 200000时，理论运行时间 = 91.071200（s）

n = 300000时，理论运行时间 = 204.910200（s）

n = 400000时，理论运行时间 = 364.284800（s）

n = 500000时，理论运行时间 = 569.195000（s）

理论效率曲线和实测效率曲线



通过观察理论效率曲线和实测效率曲线，不难发现，理论值与实际值误差较小，且图像符合O()的二次曲线。但误差随着数据规模的增大而增大，这可能是由于swap函数的调用导致程序花销随着数据规模增大所致的，数据规模越大，函数调用的次数越多，导致程序的除算法本身外的时间花销增大。

将swap函数去除，直接手写交换代码，得到的新的结果如下

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 17.082000 | 17.082000 |
| 200000 | 68.237000 | 68.328000 |
| 300000 | 153.298500 | 153.738000 |
| 400000 | 272.922000 | 273.312000 |
| 500000 | 428.435000 | 427.050000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 17.082000（s）

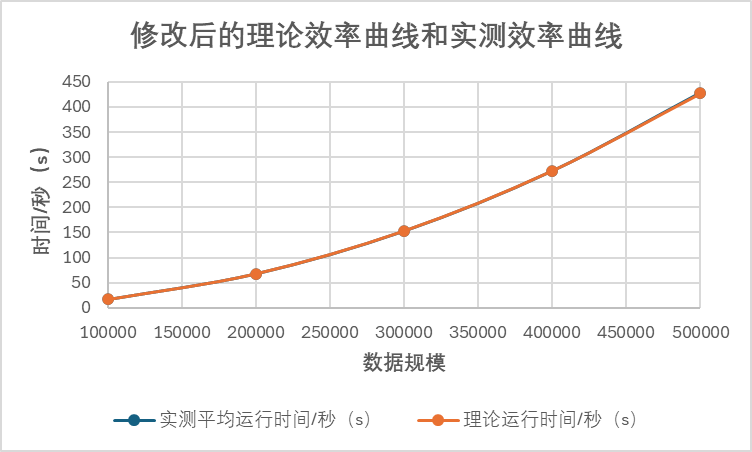
n = 200000时，理论运行时间 = 68.328000（s）

n = 300000时，理论运行时间 = 153.738000（s）

n = 400000时，理论运行时间 = 273.312000（s）

n = 500000时，理论运行时间 = 427.050000（s）

修改后的理论效率曲线和实测效率曲线



不难发现，修改后的选择排序经验分析与理论分析基本一致，图像符合O()的二次曲线，且运行时间也比修改前要少，说明调用swap函数确实对冒泡排序算法运行时间产生了较大影响。

1. **插入排序**

根据20组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 3.531200 | 3.531200 |
| 200000 | 13.974250 | 14.124800 |
| 300000 | 31.371900 | 31.780800 |
| 400000 | 55.918400 | 56.499200 |
| 500000 | 87.012450 | 88.280000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 3.531200（s）

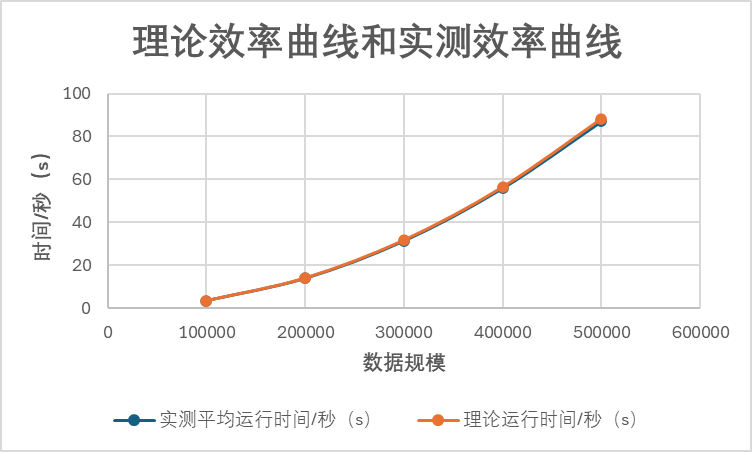
n = 200000时，理论运行时间 = 14.124800（s）

n = 300000时，理论运行时间 = 31.780800（s）

n = 400000时，理论运行时间 = 56.499200（s）

n = 500000时，理论运行时间 = 88.280000（s）

理论效率曲线和实测效率曲线



通过观察理论效率曲线和实测效率曲线，不难发现，经验分析与理论分析基本一致，理论值与实际值误差较小，且图像符合O()的二次曲线。

1. **合并排序**

根据20组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 0.044650 | 0.044650 |
| 300000 | 0.126850 | 0.146732 |
| 500000 | 0.212450 | 0.254459 |
| 1000000 | 0.429900 | 0.535800 |
| 3000000 | 1.290650 | 1.735221 |
| 5000000 | 2.146400 | 2.991090 |
| 10000000 | 4.317350 | 6.251000 |
| 30000000 | 13.288150 | 20.031210 |
| 50000000 | 22.344500 | 34.375900 |
| 100000000 | 46.036750 | 71.440000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 0.044650（s）

n = 300000时，理论运行时间 = 0.146732（s）

n = 500000时，理论运行时间 = 0.254459（s）

n = 1000000时，理论运行时间 = 0.535800（s）

n = 3000000时，理论运行时间 = 1.735221（s）

n = 5000000时，理论运行时间 = 2.991090（s）

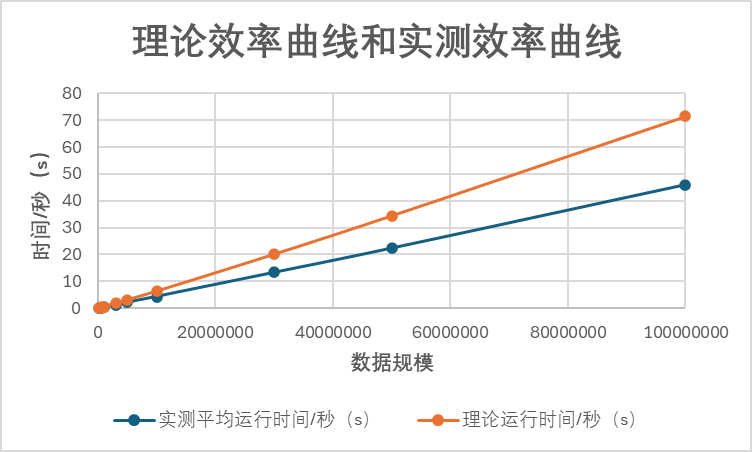
n = 10000000时，理论运行时间 = 6.251000（s）

n = 30000000时，理论运行时间 = 20.031210（s）

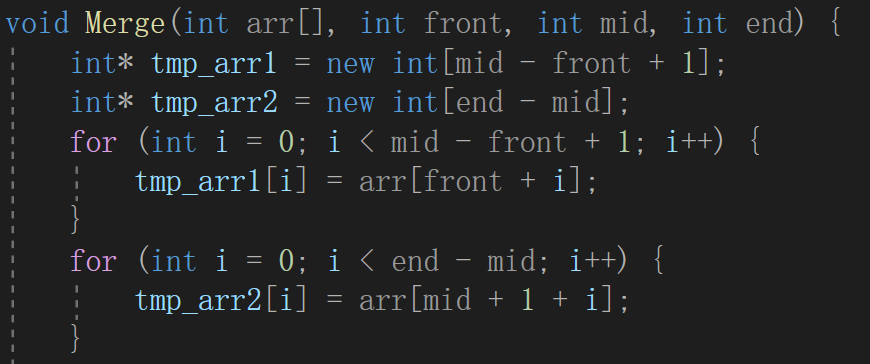
n = 50000000时，理论运行时间 = 34.375900（s）

n = 100000000时，理论运行时间 = 71.440000（s）

理论效率曲线和实测效率曲线



通过观察理论效率曲线和实测效率曲线，能发现理论值与实际值存在误差，且误差随着数据规模的增大而增大。猜测这可能是由于在合并排序内额外生成数组空间（原代码中额外生成数组空间是为了把子数组从原数组中复制下来，再合并两子数组进行排序，排序的数组放回原数组中），导致基准点的运行时间也出现了偏差，抬高了后面理论运行时间。故修改代码，让用于额外的数组空间在进入函数前就生成，在函数内直接使用该数组空间即可（在函数外便生成空间大小为n的数组，将子数组合并排序后的结果暂存在该数组，然后再把该数组排序过的范围复制回原数组），这样就无需在合并排序的函数内反复生成空间。



修改后的实测平均运行时间和理论运行时间如下

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 0.012800 | 0.012800 |
| 300000 | 0.041800 | 0.042064 |
| 500000 | 0.072900 | 0.072947 |
| 1000000 | 0.153450 | 0.153600 |
| 3000000 | 0.478650 | 0.497443 |
| 5000000 | 0.823500 | 0.857468 |
| 10000000 | 1.720300 | 1.792000 |
| 30000000 | 5.412250 | 5.742429 |
| 50000000 | 9.294800 | 9.854682 |
| 100000000 | 20.134500 | 20.480000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 0.012800（s）

n = 300000时，理论运行时间 = 0.042064（s）

n = 500000时，理论运行时间 = 0.072947（s）

n = 1000000时，理论运行时间 = 0.153600（s）

n = 3000000时，理论运行时间 = 0.497443（s）

n = 5000000时，理论运行时间 = 0.857468（s）

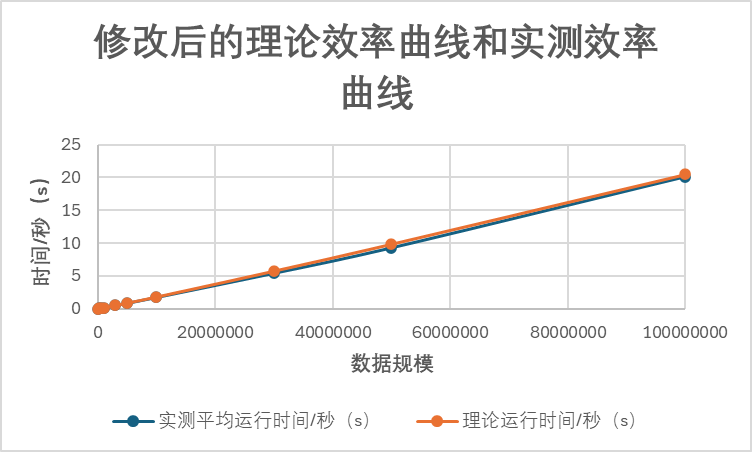
n = 10000000时，理论运行时间 = 1.792000（s）

n = 30000000时，理论运行时间 = 5.742429（s）

n = 50000000时，理论运行时间 = 9.854682（s）

n = 100000000时，理论运行时间 = 20.480000（s）

修改后的理论效率曲线和实测效率曲线



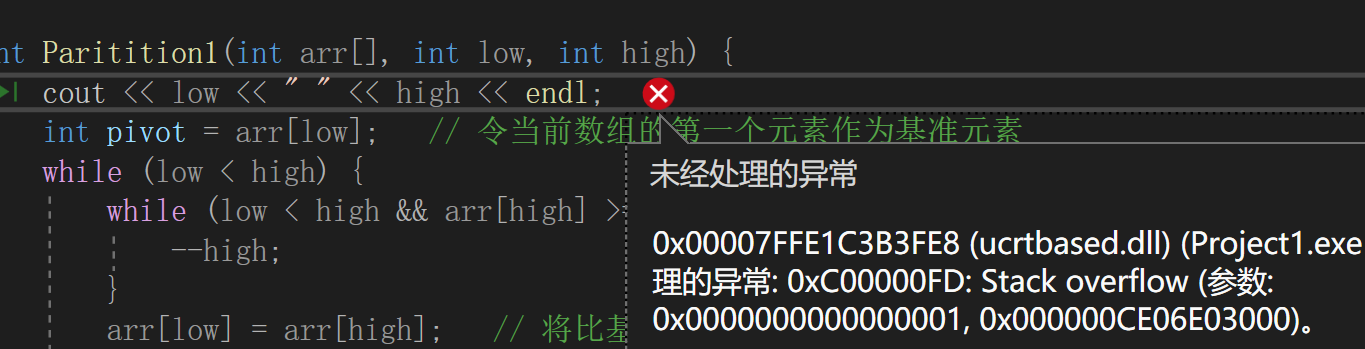
通过观察修改后的理论效率曲线和实测效率曲线，不难发现，经验分析与理论分析基本一致，理论值与实际值误差较小，且图像符合O(nlogn)的曲线，函数内不必要的数据空间生成确实影响了基准点运行时间，造成了修改前理论与实际误差较大的情况；同时，减少了不必要的空间的生成也有助于提高运行速度，减少了运行时间。

1. **快速排序**

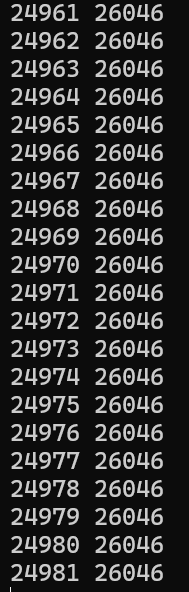
根据20组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

在运行过程中，出现了stack overflow的问题

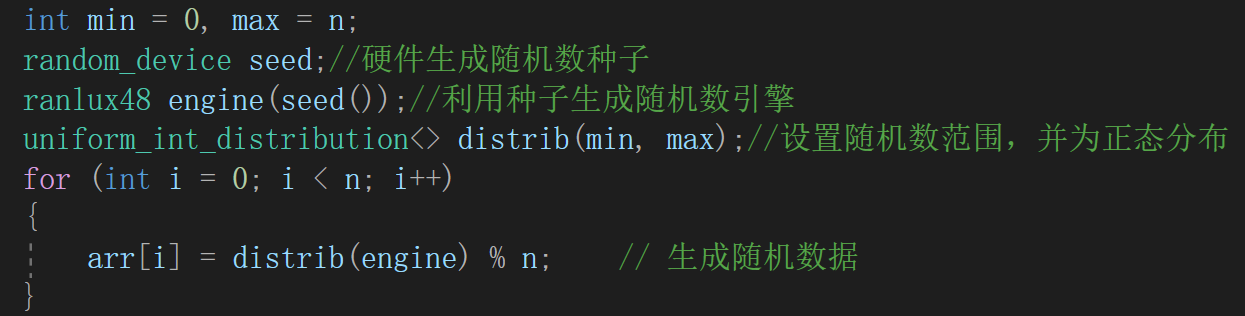


经检查发现，这是由于随机数据质量太差，多次出现有序和重复的情况，导致后面递归的划分不均匀。



上图显示的是在递归过程中打印出来的递归子数组范围，不难看出，每次递归得到的基准元素均为子数组开头的元素，这便导致数组的划分一边倒。这样不合理的划分最终导致的便是递归深度过高，导致栈爆（stack overflow）了。

于是修改随机数据生成器，将原本使用的<cstdlib>中的rand（）和srand（）函数替换成了<random>中的随机数生成引擎、随机数分布律、不确定随机数和预定义来生成随机数据，新生成的数据也就足够随机了。



根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 100000 | 0.008950 | 0.008950 |
| 300000 | 0.028300 | 0.029412 |
| 500000 | 0.048700 | 0.051006 |
| 1000000 | 0.101250 | 0.107400 |
| 3000000 | 0.325600 | 0.347821 |
| 5000000 | 0.562700 | 0.599558 |
| 10000000 | 1.157800 | 1.253000 |
| 30000000 | 3.685250 | 4.015214 |
| 50000000 | 6.572750 | 6.890578 |
| 100000000 | 13.429950 | 14.320000 |

理论运行时间的计算：

n = 100000时，理论运行时间 = 0.008950（s）

n = 300000时，理论运行时间 = 0.029412（s）

n = 500000时，理论运行时间 = 0.051006（s）

n = 1000000时，理论运行时间 = 0.107400（s）

n = 3000000时，理论运行时间 = 0.347821（s）

n = 5000000时，理论运行时间 = 0.599558（s）

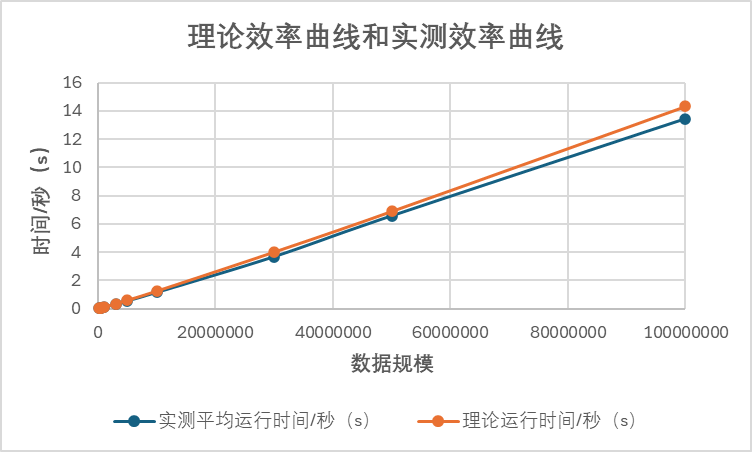
n = 10000000时，理论运行时间 = 1.253000（s）

n = 30000000时，理论运行时间 = 4.015214（s）

n = 50000000时，理论运行时间 = 6.890578（s）

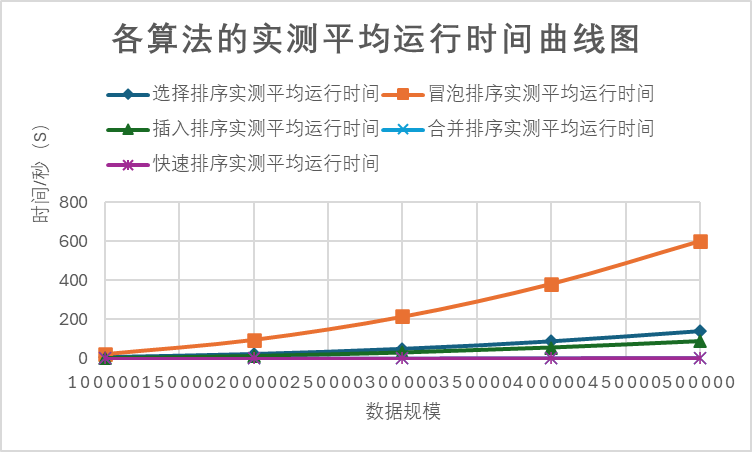
n = 100000000时，理论运行时间 = 14.320000（s）

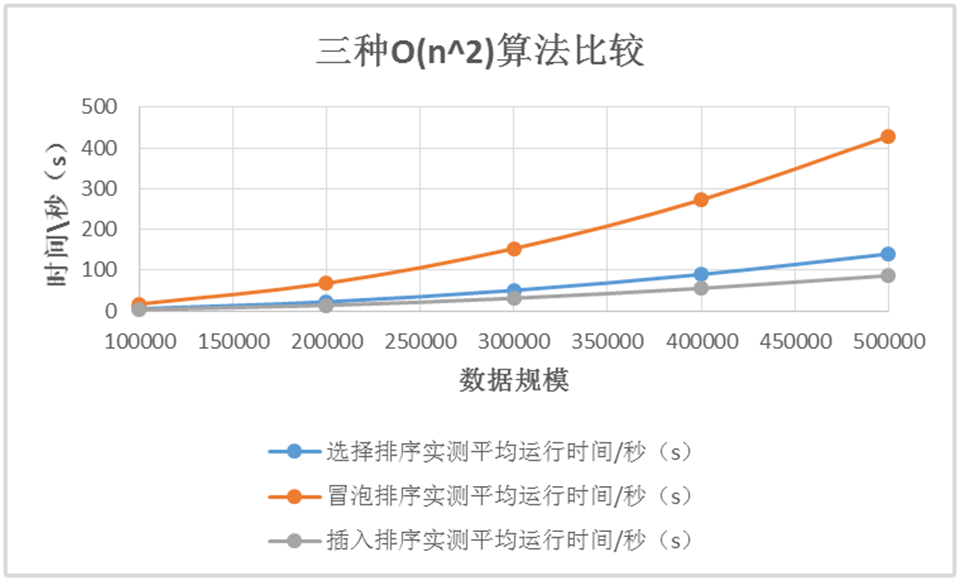
理论效率曲线和实测效率曲线

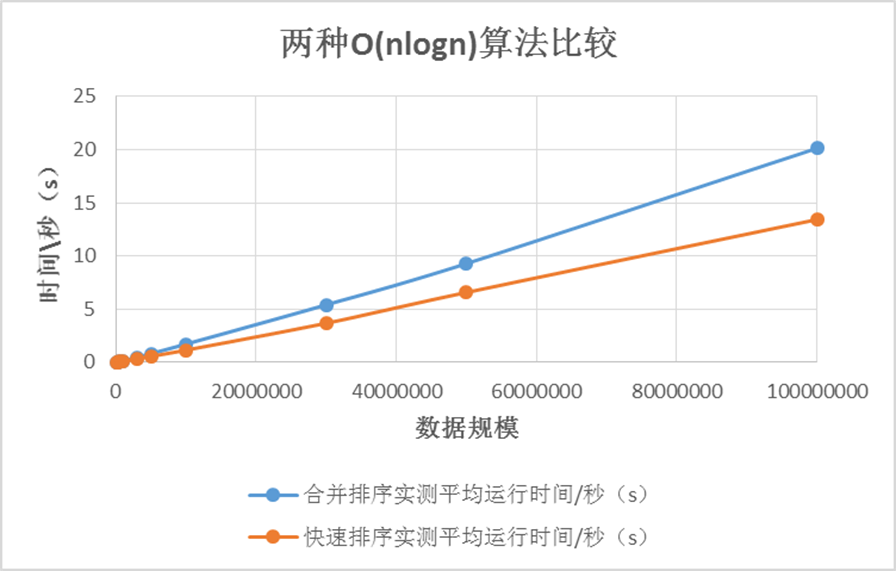


通过观察理论效率曲线和实测效率曲线，不难发现，经验分析与理论分析基本一致，理论值与实际值误差较小，且图像符合O(nlogn)的曲线。

1. **各排序算法实测运行时间曲线图**







通过观察曲线图不难看出，各排序算法在数据规模100000到500000对数据的排序效率从高到低的排名为：快速排序≈合并排序>插入排序>选择排序>冒泡排序。这种排序效率的排名是由算法的时间复杂度和内部实现特点所决定的。

快速排序和合并排序：它们的时间复杂度都是 O()。在大规模数据集上，它们通常是最快的排序算法之一。快速排序具有良好的平均性能，尤其在数据集比较均匀的情况下。而合并排序则是一种稳定的排序算法，对于大规模数据集也有很好的性能表现。对数据规模100000到500000的数组排序对于它们来说属于一秒内就能完成的任务。

插入排序：它的时间复杂度为 O()，在小规模数据集上通常比较有效，因为它的常数因子较小。

选择排序：它的时间复杂度也是 O()，它的性能比插入排序差一些，可能是由于在排序过程中存在大量的元素交换操作，使得在大规模数据集上性能表现较差。

冒泡排序：与选择排序类似，冒泡排序的时间复杂度也是 O()，并且它也存在大量的元素交换操作。由于其性能较差，因此在实际应用中很少被使用。

**问题2：**

假设数据规模为n，n分别取值为1000000,3000000,5000000, 10000000,30000000,50000000,100000000, 300000000, 500000000, 1000000000，记录下三种算法对10组随机测试样本选取最大的10个数的实测平均运行时间，得到数据如下。

1. **最大堆算法**

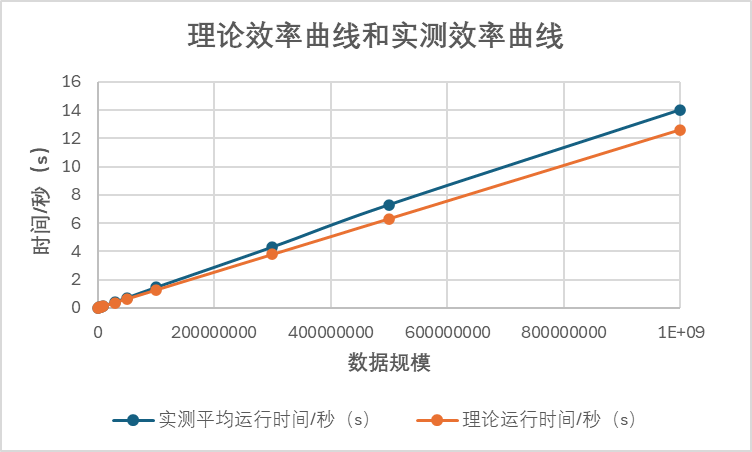
根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 1000000 | 0.012600 | 0.012600 |
| 3000000 | 0.038400 | 0.037800 |
| 5000000 | 0.067600 | 0.063000 |
| 10000000 | 0.137800 | 0.126000 |
| 30000000 | 0.416200 | 0.378000 |
| 50000000 | 0.702800 | 0.630000 |
| 100000000 | 1.446400 | 1.260000 |
| 300000000 | 4.307800 | 3.780000 |
| 500000000 | 7.297000 | 6.300000 |
| 1000000000 | 14.012100 | 12.600000 |

理论效率曲线和实测效率曲线



通过观察堆排序运行时间曲线图，可以看到理论与实测效率误差较小，最大堆算法时间曲线图基本符合O(n)的曲线，在10亿的数据中挑选出最大的十个数的所用时间为16.979200s。误差随着数据规模的增大而增大，这可能是由于堆排序中存在多次调用堆调整函数所导致的，由此增加了实测时间。

1. **基于选择排序的算法**

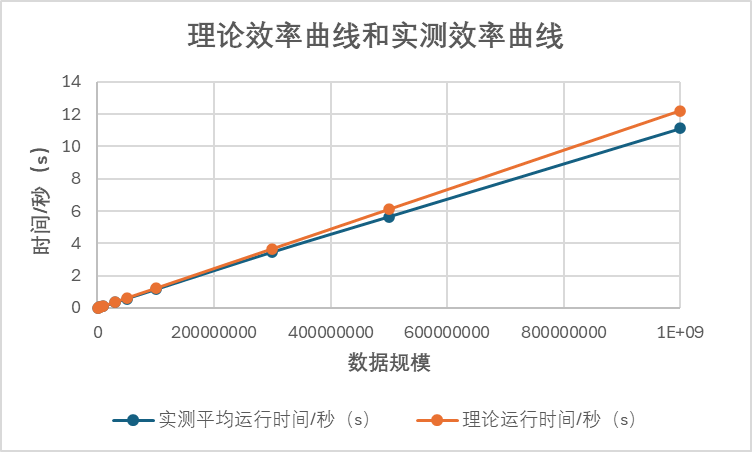
根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 1000000 | 0.012200 | 0.012200 |
| 3000000 | 0.035200 | 0.036600 |
| 5000000 | 0.056500 | 0.061000 |
| 10000000 | 0.114600 | 0.122000 |
| 30000000 | 0.338700 | 0.366000 |
| 50000000 | 0.574400 | 0.610000 |
| 100000000 | 1.149500 | 1.220000 |
| 300000000 | 3.467600 | 3.660000 |
| 500000000 | 5.655500 | 6.100000 |
| 1000000000 | 11.121300 | 12.200000 |

理论效率曲线和实测效率曲线



通过观察选择排序运行时间曲线图，可以看到理论与实测效率误差较小，基于选择排序的算法实测平均运行时间曲线图基本符合O(n)的曲线，在10亿的数据中挑选出最大的十个数的所用时间为11.121300s。

1. **基于快速排序的算法**

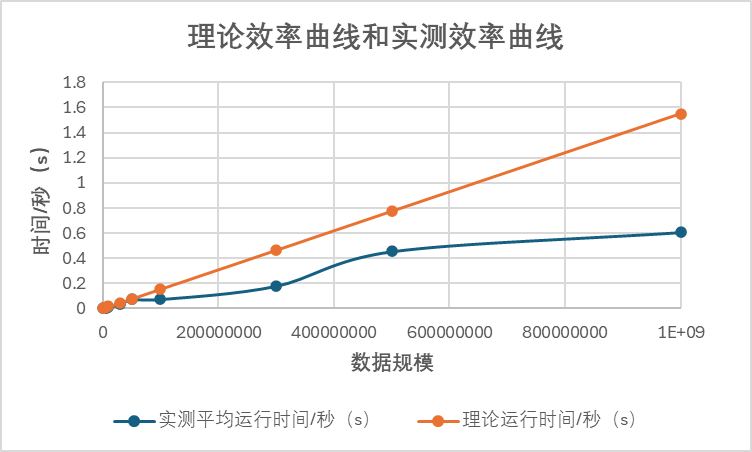
根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 1000000 | 0.001550 | 0.001550 |
| 3000000 | 0.003950 | 0.004650 |
| 5000000 | 0.005800 | 0.007750 |
| 10000000 | 0.013300 | 0.015500 |
| 30000000 | 0.033250 | 0.046500 |
| 50000000 | 0.073150 | 0.077500 |
| 100000000 | 0.073600 | 0.155000 |
| 300000000 | 0.179250 | 0.465000 |
| 500000000 | 0.455250 | 0.775000 |
| 1000000000 | 0.608350 | 1.550000 |

理论效率曲线和实测效率曲线



通过观察选择排序运行时间曲线图，可以看到基于快速排序的算法实测平均运行时间曲线图与O(n)的曲线有偏差，运行时间随数据规模的增大会出现抖动，而非像O(n)曲线一样线性增大。这可能是由于快速排序的不稳定性所导致的，快速排序的效率受数据质量影响较大，越是随机无序或近似随机无序，数据划分越平均，快速排序越快，数据已经完全有序（或接近有序），数据划分越不平均，快速排序越慢。由此，因为测试的数据组太少，得到的平均时间还不能体现该算法真正的平均水平，所以导致理论与实测的误差过大。在10亿的数据中挑选出最大的十个数的所用时间为7.754400s。

1. **最小堆算法**

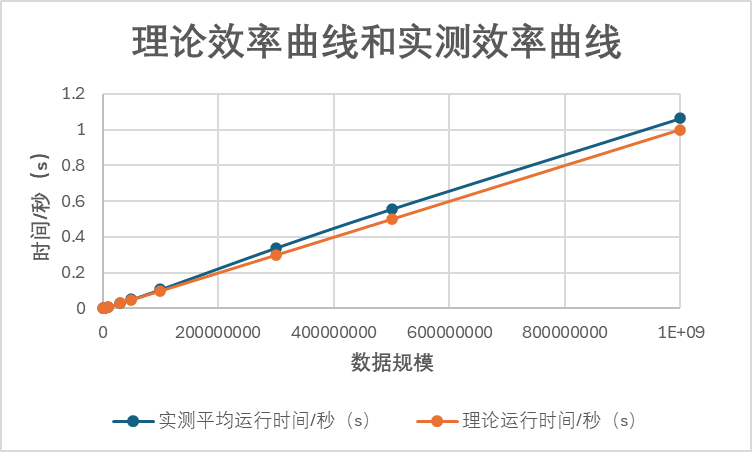
根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

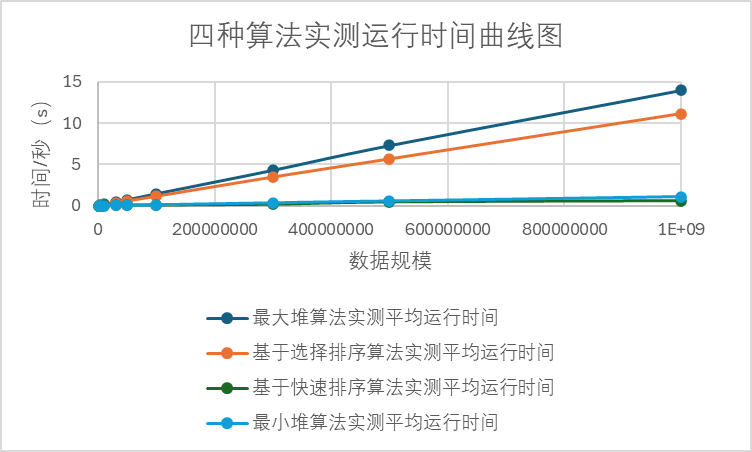
|  |  |  |
| --- | --- | --- |
| 数据规模n | 实测平均运行时间/秒（s） | 理论运行时间/秒（s） |
| 1000000 | 0.001000 | 0.001000 |
| 3000000 | 0.002800 | 0.003000 |
| 5000000 | 0.004800 | 0.005000 |
| 10000000 | 0.009800 | 0.010000 |
| 30000000 | 0.030800 | 0.030000 |
| 50000000 | 0.051300 | 0.050000 |
| 100000000 | 0.106500 | 0.100000 |
| 300000000 | 0.339100 | 0.300000 |
| 500000000 | 0.555800 | 0.500000 |
| 1000000000 | 1.063000 | 1.000000 |

理论效率曲线和实测效率曲线



通过观察堆排序运行时间曲线图，可以看到理论与实测效率误差较小，最小堆算法实测平均运行时间曲线图基本符合O(n)的曲线，在10亿的数据中挑选出最大的十个数的所用时间为1.063000s。

1. **四种算法实测运行时间曲线图**



通过观察四种改良算法实测运行时间曲线图，不难发现，四种排序算法在数据规模1000000到1000000000挑选出最大的十个数的效率从高到低的排名为：最小堆算法≈基于快速排序算法>基于选择排序算法>最大堆算法。

虽然四种算法的时间复杂度都到了O(n)级别，但实际运行时间还是略有差别，最大堆算法在建立堆和堆调整需要调用很多次函数，这可能是其性能稍逊于基于选择排序算法的原因；最小堆算法则依靠其维护一个长度为k的最小堆的特性，将时间和空间复杂度都降到了一个很小的水平，运行时间上直逼基于快速排序算法；基于快速排序算法本身性能就很优越，再加之优化后能及时退出函数，避免了不必要的递归，使得其成为该问题运行速度最快的算法，

1. **经验总结**

在解决问题一中，会出现随机数据的质量过差导致了理论运行时间与实测运行时间相差太大，理论与实际不匹配的形况，最为明显的就是快速排序。原本一开始使用<cstdlib>中的rand（）和srand（）函数来生成随机数，但查阅过资料发现，<cstdlib>中的rand（）和srand（）函数是C语言使用的随机数生成方法，通过线性同余法计算，然而rand（）不能保证所生成序列的质量，在随机性、统计分布性质和序列的周期上有很大的缺陷，不能满足用于科学研究的严肃随机数生成。因此，我改用了<random>中的随机数生成引擎、随机数分布律、不确定随机数和预定义来进行随机数生成，生成的随机数质量比之前要好，同时快速排序也能正常运行。可见，排序算法的效率不仅仅受自身算法设计的影响，随机数据的质量也会对其效率产生作用。

在解决问题二中，考虑问题的角度是如何利用排序算法在挑选出最大的十个数后就能跳出程序，尽早结束进程以提升运行速度。最大堆堆调整的特性非常适合本问题，只需要10次把堆调整成最大堆后，把堆顶元素筛选出来即可，这样只需要十次循环操作便可解决该问题进而退出进程。选择排序同样也适合本问题，只需要十次循环在数组中挑选出最大的十个数就能结束进程。快速排序则需要额外的代码对结果进行检测，检测到数组最右端长度为十的数据段被递归排序后才结束进程。可见，利用最大堆、选择排序的算法本身的特性可以解决问题，但同样也可以借助检测等手段来使得快速排序这样速度快的算法得以应用。

同时，在分析一个算法的优点时，不能仅关注其运行速度，还得关注其现实的可行性。在现实生活中，计算机在面对海量的数据时不一定都能直接存入内存中，因此最大堆算法、基于选择排序的算法、基于快速排序的算法都可能由于内存不够而无法在海量数据中挑选出最大的k个数；而最小堆算法恰好可以很好地应对该问题，其在内存中只需要维护一个大小为k的最小堆，当其他数据到来时，可以一个一个地处理，无需全部直接存入内存中，进而可以在海量数据中挑选出最大的k个数，突破了内存的限制。因此，衡量一个算法的好坏不仅仅看其运行的速度，还需看其现实可行性。

|  |
| --- |
| 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。