

华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析

年级：22 级

上机实践成绩：

指导教师：金澈清

姓名：石季凡

上机实践名称：顺序统计量

学号：

上机实践日期：2023.4.4

10225501403

上机实践编号：No.5

组号：1-403

一、目的

1. 熟悉算法设计的基本思想
2. 掌握随机选择算法（rand select）的方法
3. 掌握选择算法（SELECT）的方法

二、实验内容

1. 编写随机选择算法和 SELECT 算法；
2. 随机生成 $1e2$ 、 $1e3$ 、 $1e4$ 、 $1e5$ 、 $1e6$ 个数，使用随机选择算法和 SELECT 算法找到第 $0.5N$ 大的数输出，并画图描述不同情况下的运行时间差异；
3. 随机生成 $1e6$ 个数，使用随机选择算法和 SELECT 算法找到第 $0.2N$ 、 $0.4N$ 、 $0.6N$ 、 $0.8N$ 大的数输出，并画图描述不同情况下的运行时间差异；
4. 递增生成 $1e2$ 、 $1e3$ 、 $1e4$ 、 $1e5$ 、 $1e6$ 个数，使用随机选择算法和 SELECT 算法找到第 $0.5N$ 大的数输出，并画图描述不同情况下的运行时间差异；
5. 随机生成 $1e2$ 、 $1e3$ 、 $1e4$ 、 $1e5$ 、 $1e6$ 个数，使用 merge sort 找到第 $0.5N$ 大的数输出，并画图描述不同情况下的运行时间差异；
6. 对比随机选择算法和 SELECT 算法以及 merge sort。

三、使用环境

C/C++集成编译环境。

Windows11

注：代码分为优化前和优化后，报告内必须完成的实验均为优化前代码完成。

四、实验过程。

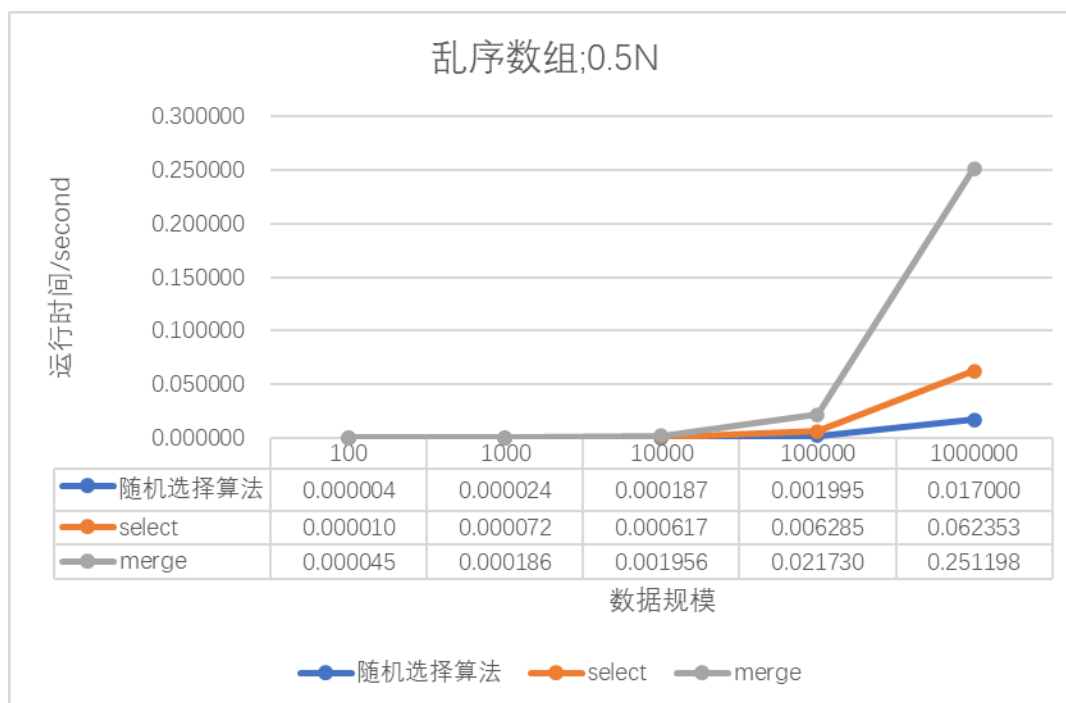
首先，我分别实现了随机选择算法与 SELECT 算法，接下来进行实验

注：SELECT 默认分组为 5，若修改则会提出；实验数据默认范围为 $1 \sim 10N$

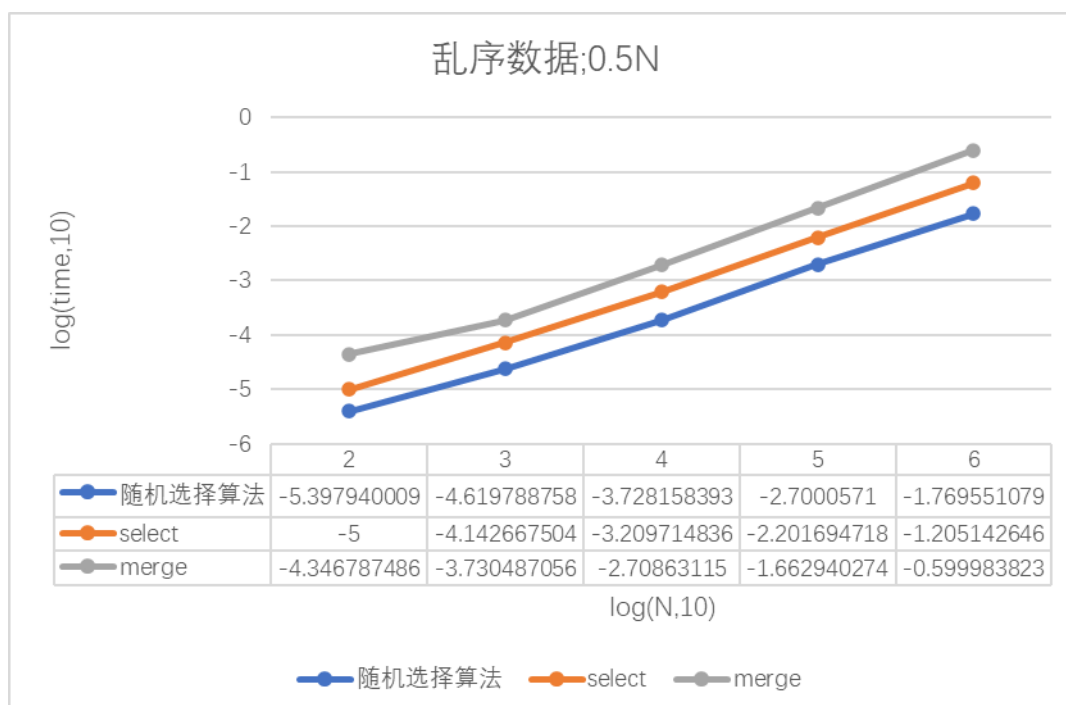
实验内容 2&5&6

数据以及结果如下，由于原始数据不易观察，我对运行时间与数据规模均取了以 10 为底的对数。

下面是对数图



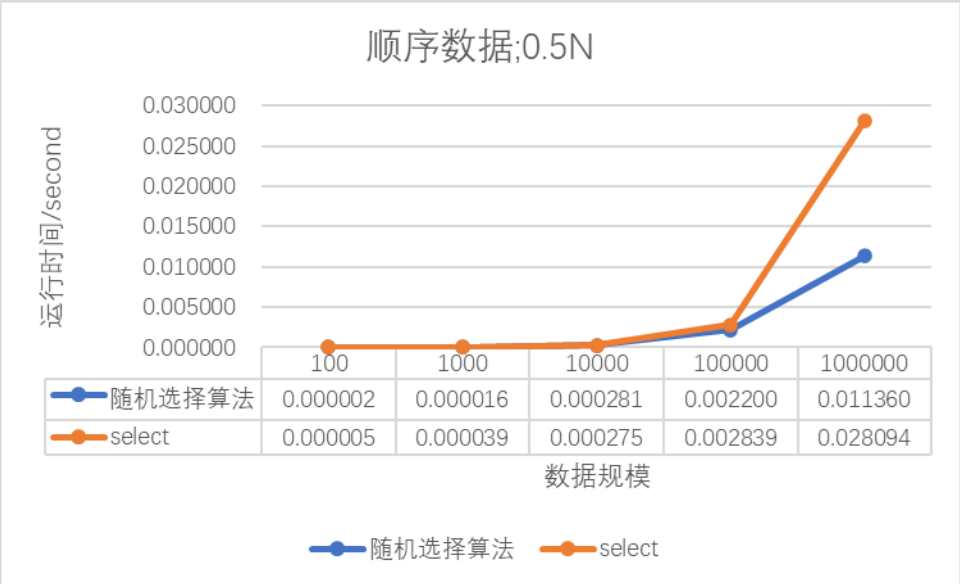
下面把所有数据取以 10 为底的对数



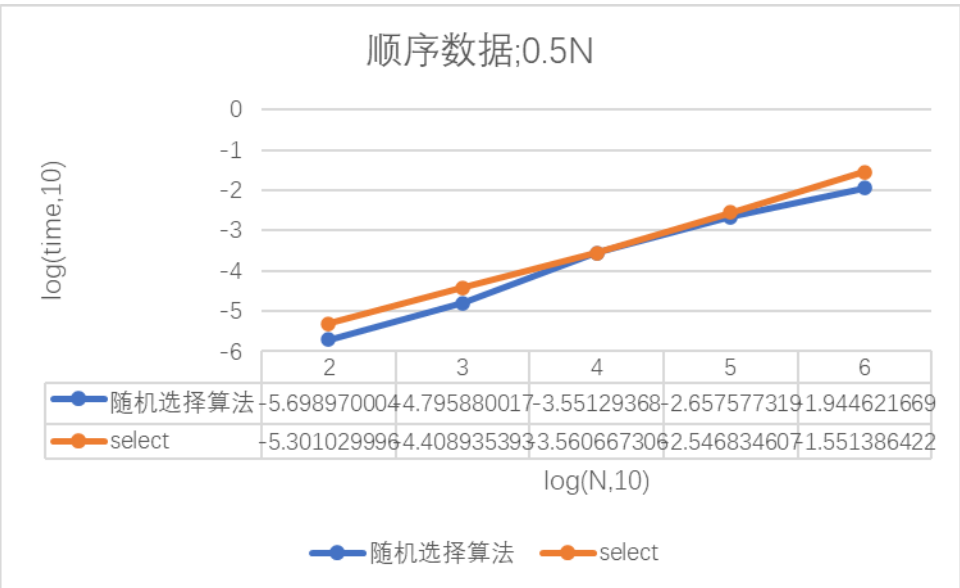
可以看到，在实验的规模的数据下，随机选择算法完全优于 **SELECT** 算法，但是随机选择算法有小规模的波动，随机选择算法则更为稳定，这主要由于 **SELECT** 算法的最坏情况为线性，而随机选择算法的期望为线性，最坏情况的时间复杂度为 $O(n^2)$ 。

对于归并排序，由于其必须完全排序完才能完成，而且其为比较排序算法，时间下限为 $n \log n$ ，因此远不及其他两种算法。

实验内容 4

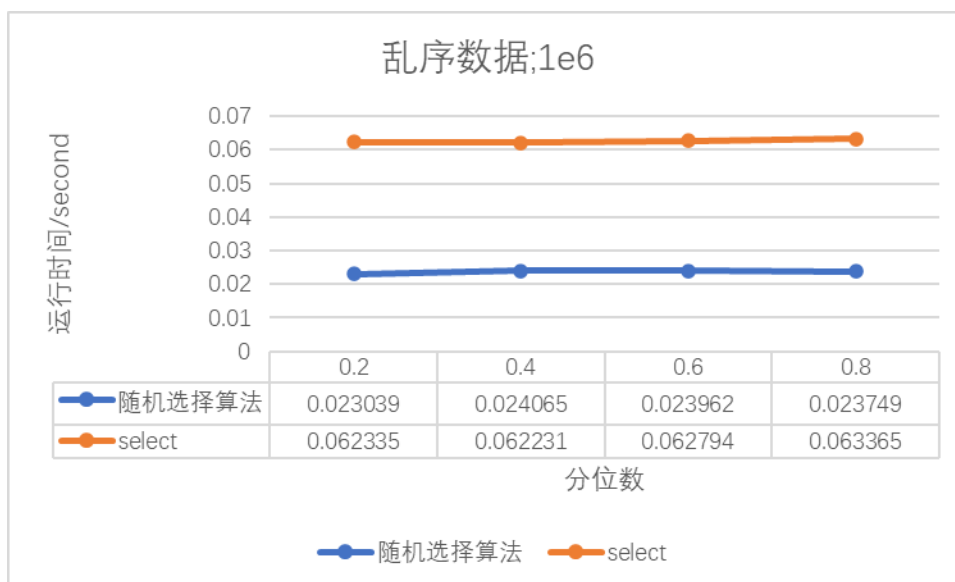


下面为对数据取以 10 为底的对数



可以看到，随机选择算法仍然有着微小波动。

实验内容3



通过这些数据，我们可以发现，这两种算法均较为稳定，时间几乎无差异。


五、总结

进行完上述的实验内容，我对于 **SELECT** 算法与随机选择算法的关系有一个思考，**SELECT** 算法非常稳定，最坏情况为线性，但是线性的系数很大，对于随机选择算法，虽然经过一次随机，但是还是有一定可能出现分组白忙活的情况，那么我们能不能将二者结合呢？

我的想法是，重新设计一个函数，我称之为 **mix_select**，其主体为随机选择算法，但是在进行递归时，我定义了一个新的变量 **index**，用其来衡量 **partition** 后所得的 **middle** 的位置，即 $\text{index} = (\text{left} - \text{middle}) / (\text{right} - \text{left} + 1)$ ，个人主观判断当这个指数大于 95% 或者小于 5% 时，说明这次的分组是较为不幸的，不幸的出现有两种原因，一位数据不利于分组，二为随机出的分组点不幸运，但是第二种可能性主观看来可能性较小，所以我们可以猜测在这个分组中数据不利于分组，此时可以调用 **SELECT** 算法来进行，然后在 **SELECT** 中也引入 **index** 进行检测，直到 **index** 处于 3%~97%。

但是这样做仍然不够严谨，毕竟主观的猜测太多，因此我又引入了一个函数参数 **flag**，每次 **index** 过高或过低时令 **flag++**，如果没有就赋值为 0，**flag** 达到阈值（阈值为 5，即 6 次）时再调用 **SELECT** 算法，**SELECT** 算法内也引入 **flag**，一直递归，直到 **index** 正常，赋值 **flag** 为 0，转为递归 **mixselect**，于是乎，修改部分的代码如

首先是 mixselect 算法的特殊



```
C test_4.c 1 X 设置
C test_4.c > main()
218 void mixselect(int nums[], int left, int right, int target, int flag)
219 {
220     srand((unsigned)time(NULL) + rand());
221     if (right - left > 1)
222     {
223         int tmp = (double)rand() / RAND_MAX * (right - left) + left + 1;
224         int max = right, min = left;
225         if (nums[max] < nums[min])
226         {
227             max = left, min = right;
228         }
229         if (nums[tmp] <= nums[max] && nums[tmp] >= nums[min])
230             swap(&nums[tmp], &nums[left]);
231         else if (nums[tmp] > nums[max])
232             swap(&nums[max], &nums[left]);
233         else
234             swap(nums + min, nums + left);
235     }
236     int middle = partition(nums, left, right);
237     if (middle == target)
238         return;
239     double index = (double)(left - middle) / (double)(right - left + 1);
240     if (index < 0.03 || index > 0.97)
241     {
242         flag++;
243         if (flag == 5)
244         {
245             if (middle > target)
246                 my_select(nums, left, middle - 1, target, flag);
247             else
248                 my_select(nums, middle + 1, right, target, flag);
249         }
250         else
251         {
252             if (middle > target)
253                 mixselect(nums, left, middle - 1, target, flag);
254             else
255                 mixselect(nums, middle + 1, right, target, flag);
256         }
257     }
258     else
259     {
260         flag=0;
261         if (middle > target)
262             mixselect(nums, left, middle - 1, target, 0);
263         else
264             mixselect(nums, middle + 1, right, target, 0);
265     }
266 }
267 // int main()
```

然后是对 SELECT(my_select)函数修改部分

```
int middle = partition(nums, left, right);
double index = (double)(left - middle) / (double)(right - left + 1);
if (middle == target)
    return;
if (index < 0.03 || index > 0.97)
{
    if (middle > target)
        my_select(nums, left, middle - 1, target, flag);
    else
        my_select(nums, middle + 1, right, target, flag);
}
else
{
    if (middle > target)
        mixselect(nums, left, middle - 1, target, 0);
    else
        mixselect(nums, middle + 1, right, target, 0);
}
```

那么这样修改后有效吗？

我们来用数据验证一下

首先是水杉 oj 进行验证

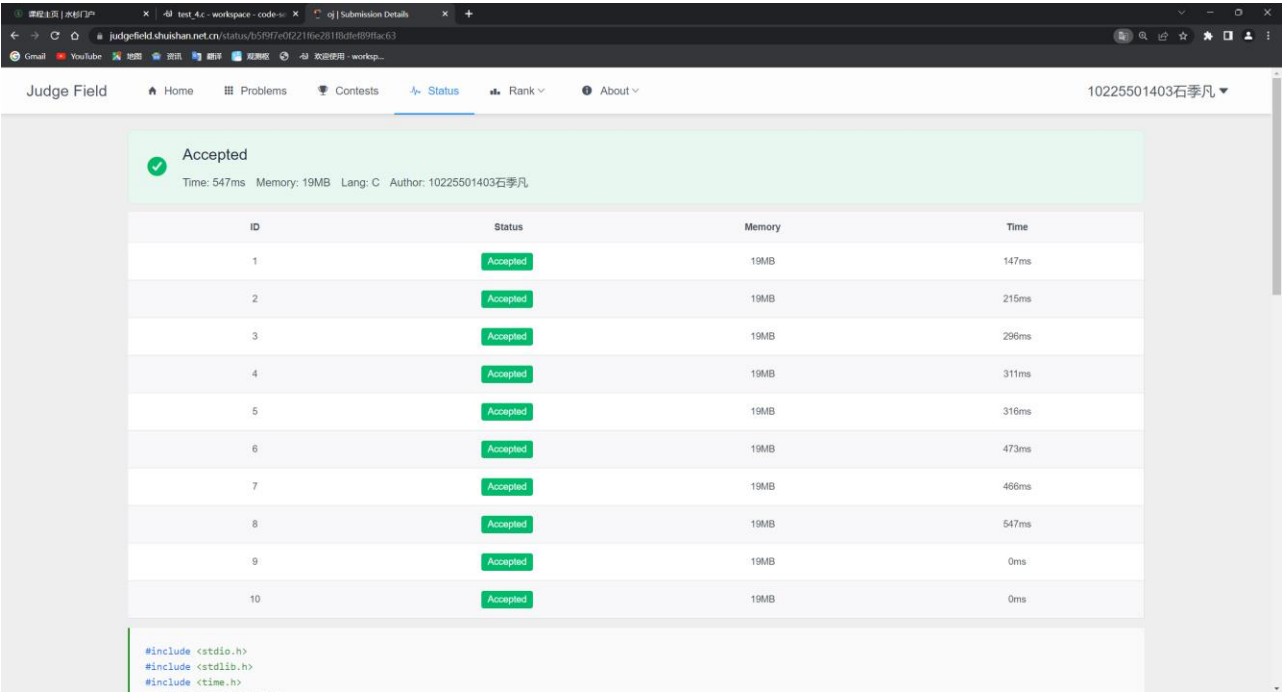
The screenshot shows the Judge Field submission details page. At the top, it indicates the submission is "Accepted" with a green checkmark. Below this, a table displays the results of 10 test cases. Each row shows the ID, Status (Accepted), Memory (19MB), and Time (various values). The bottom of the page shows the source code, which includes standard C headers and a function definition.

ID	Status	Memory	Time
1	Accepted	19MB	150ms
2	Accepted	19MB	230ms
3	Accepted	19MB	314ms
4	Accepted	19MB	379ms
5	Accepted	19MB	377ms
6	Accepted	19MB	379ms
7	Accepted	19MB	615ms
8	Accepted	19MB	535ms
9	Accepted	19MB	0ms
10	Accepted	19MB	0ms

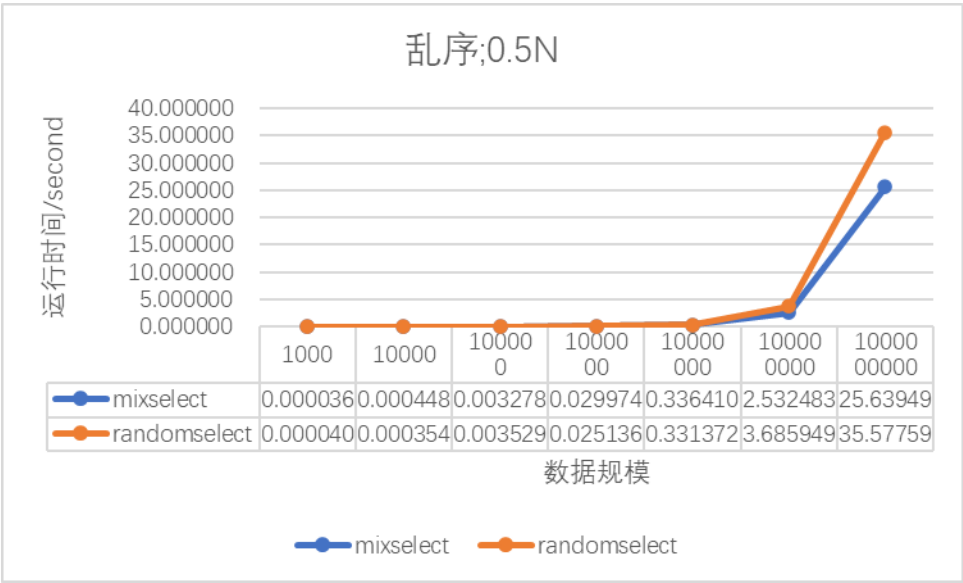
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

很显然通过了测试

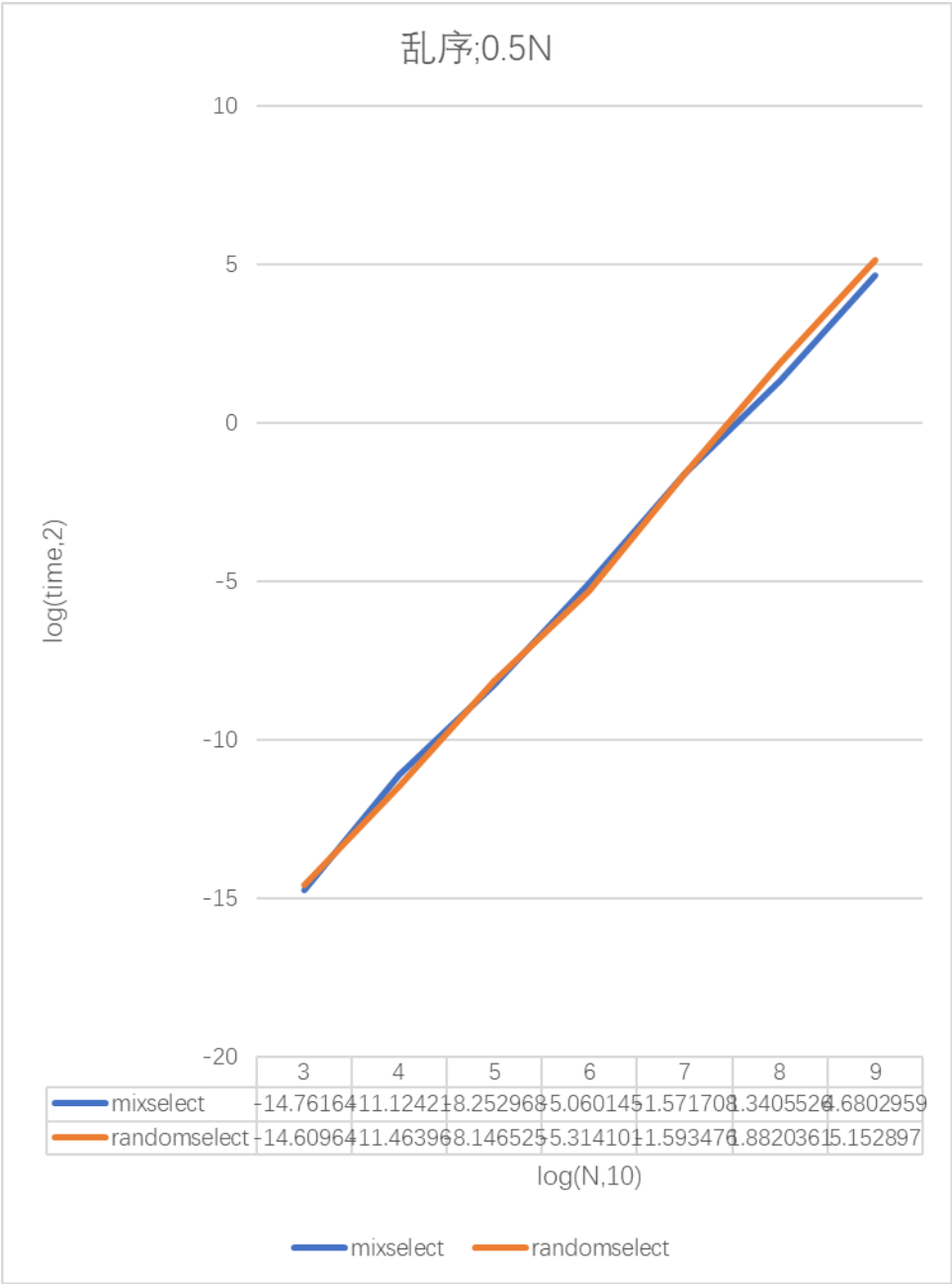
和随机选择算法对比一下



速度确实有提升，但是基本可以忽略，下面来进行实测，我们选取数据量 N 为 $1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9$ ，范围为 $10 \times N$ ，求取中位数，乱序，来进行测试这是原始数据图



这是对数图



有二者都有一定的随机性，因此每组数据均进行了多次取平均值，同时为了确保二者的结合是有效的，我定义了一个 `times` 用来记录 `SELECT` 是否被调用，其中一次的数据如图

```
请依次输入数据量、数据最小值、数据最大值、数据类型，我们规定，1为顺序，-1为逆序，0为乱序
100000000 1 1000000000 0
混合选择选择耗时: 2.388556 秒
随机选择选择耗时: 2.753578 秒
5917
abc@9155890764d6: ~/workspace/算法基础/第五次作业$
```

可见被调用了 5971 次

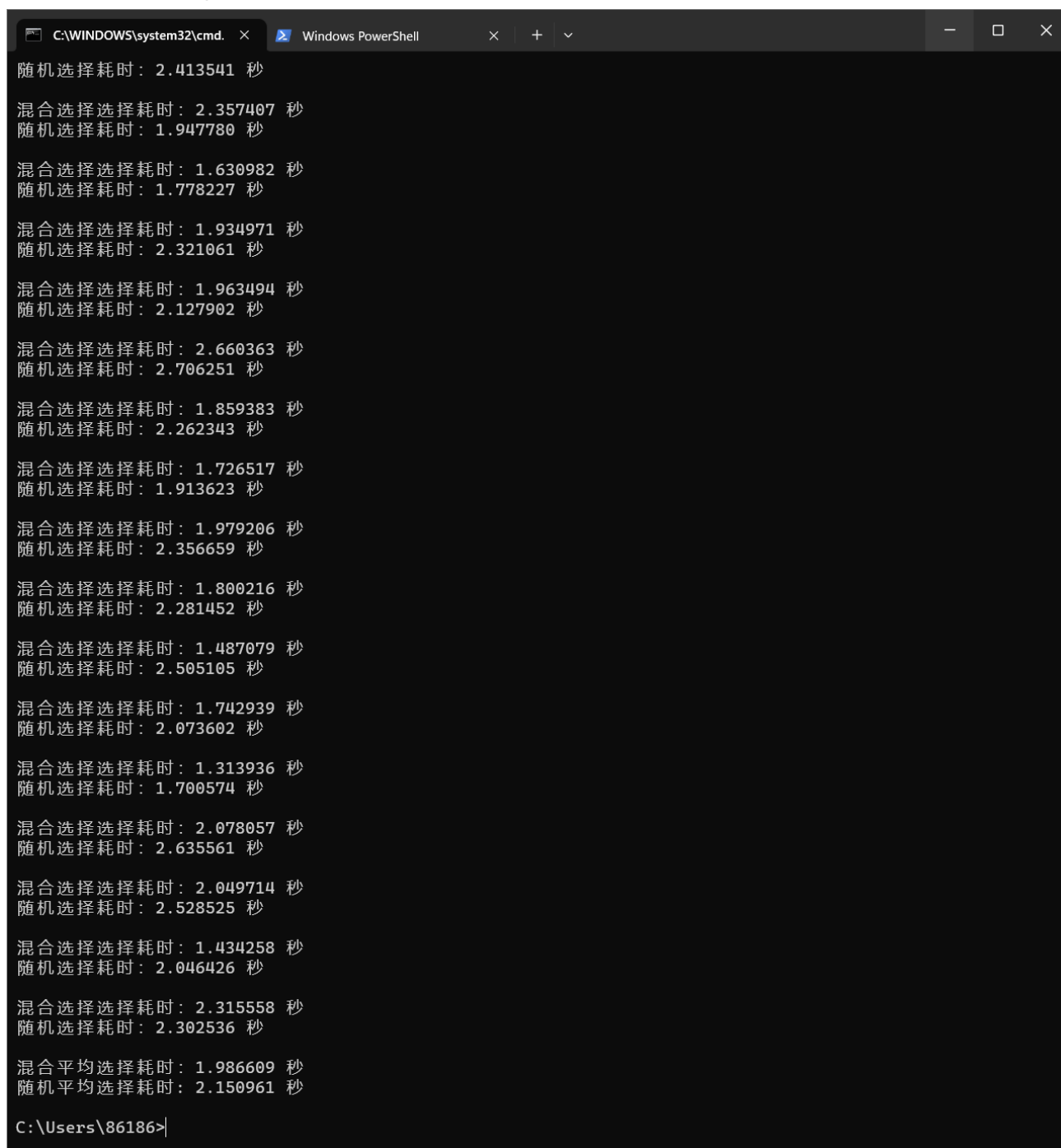
而根据上面的数据也可以知道，结合是有效的，但是实际下来随机性还是太大，二者在确定次数的实验里无法稳定保证一方快于另外一方，并且在小数量级是无明显差距，所以

目前实现的算法实用意义不大。真正有意义的是可以对于每次分组的趋势进行分析，而不是单纯的靠运气。

但是后来我又发现，通过 `times` 对过程进行分析，在 $1e8$ 这个量级时（这个量级足够大且每次运行时间较短），`times` 只要为大于 1000 时，`mixselect` 基本都是快于 `randomselect`，这说明我们的优化其实是有效果的，同时我发现，在 $1e8$ 量级时，`flag` 阈值设为 26 时，`mixselect` 算法居然惊人的基本快于 `randomselect`，至此我找到了 $1e8$ 的一组自认为较为良好的阈值，即 `index` 的两端分别为 5%~95%，`flag` 阈值为 25（26 次）。

下面贴上 $1e8$ 时用上面提及的良好阈值时运算所得的结果（为被包含至最开始的实验）注：每组第三行为 `times` 的值

下图为 `flag` 阈值为 25，一共运行了 1000 组数据求取平均值



```
C:\WINDOWS\system32\cmd. x Windows PowerShell

随机选择耗时：2.413541 秒

混合选择选择耗时：2.357407 秒
随机选择耗时：1.947780 秒

混合选择选择耗时：1.630982 秒
随机选择耗时：1.778227 秒

混合选择选择耗时：1.934971 秒
随机选择耗时：2.321061 秒

混合选择选择耗时：1.963494 秒
随机选择耗时：2.127902 秒

混合选择选择耗时：2.660363 秒
随机选择耗时：2.706251 秒

混合选择选择耗时：1.859383 秒
随机选择耗时：2.262343 秒

混合选择选择耗时：1.726517 秒
随机选择耗时：1.913623 秒

混合选择选择耗时：1.979206 秒
随机选择耗时：2.356659 秒

混合选择选择耗时：1.800216 秒
随机选择耗时：2.281452 秒

混合选择选择耗时：1.487079 秒
随机选择耗时：2.505105 秒

混合选择选择耗时：1.742939 秒
随机选择耗时：2.073602 秒

混合选择选择耗时：1.313936 秒
随机选择耗时：1.700574 秒

混合选择选择耗时：2.078057 秒
随机选择耗时：2.635561 秒

混合选择选择耗时：2.049714 秒
随机选择耗时：2.528525 秒

混合选择选择耗时：1.434258 秒
随机选择耗时：2.046426 秒

混合选择选择耗时：2.315558 秒
随机选择耗时：2.302536 秒

混合平均选择耗时：1.986609 秒
随机平均选择耗时：2.150961 秒

C:\Users\86186>
```

经过计算可得混合选择用时相对于随机选择耗时减少了 8.3%。

下面我通过不断地尝试找到了可用的 flag 阈值

	Flag_threshold
1e3	7
1e4	10
1e5	14
1e6	16
1e7	19
1e8	25

但是实际测试下来也需要根据生成的随机数的特点来修改，以上数据的实验平台为 Windows11；16g ram；CPU Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz

至此可以有一个优化的总结。

SELECT 算法可以与 randomselect 结合，组合为更为快的新的选择函数，但是其需要对于数据有着精准的检测，检测得越精准，选择则会越节约时间，而难点也在于如何找到合适的阈值，或者说是合适的转换时机，从而达到二者完美的结合。

接下来进行本次实验的错误总结。

第一点为对于 SELECT 函数最初有误解，即对每组的中位数取中位数时，错误地调用了 get_mid 函数，正确方法是应该使用 SELECT 函数，二者相互递归调用，才能保证取得的中间值的足够幸运。

第二点为 partition 函数的设置。

这一点错误存在了很多次实验，直到这次才找到原因，即在排序顺序数据时极为慢，原因是分割时是取左侧为基准值，且是从小到大排序，所以在排序顺序数据时会非常麻烦。

第三点还是内存泄漏，经检查后修改解决。

至此，实验 5 结束。