三种排序算法的比较

MyriadDreamin 2017211279 2017211301

目录

0.1 约定

0.1.1 数组和数组的长度

$A[1...n] = [A[1]...A[n]](n >= 1)$ 总是表示一个数组,$len[A] = n$ 总是表示一个数组的长度.

0.1.2 数组切片

若 $A[l...r]$ 出现 $l > r$, 则 $A[l...r] = \varnothing$, 否则 $A[l...r] = [A[l], \ldots, A[r]]$.

0.1.3 时间复杂度表示

我们用 $T(n)$ 表示某一次特定的数据集 $\mathcal{S}$ 该算法的时间复杂度, 如无指定, 则 $T(n)$ 表示一般情况的时间复杂度.

0.1.4 对数表示

$\log n := \log_2 n, \lg n := \log_{10} n, \ln n := \log_e n.$

0.1.5 测试环境

CPU 主频 3.2GHz, 程序限制使用 10% 的 CPU 资源. 内存 16G, 测试系统为 windows10x64.

0.1.6 源代码

GitHub:https://github.com/Myriad-Dreamin/Sort-Comparsion

1 插入排序 InsertionSort

对于 $[1...i](i \leqslant n)$ 将 $A[i]$ 从后往前依次挪动找到一个恰保证有序的位置，视为结束一次插入.

1.1 正确性

采用数学归纳法.

假设:

定义 Insert$(x, A[l...r])$ 为插入操作, 它不破坏 $A[l...r]$ 之间的相对关系, 并将 $A[l...r]$ 分为 $A_{lo} = A[l...x-1], A[x], A_{hi} = A[x+1...r]$, 满足 $\forall y \in A_{lo}, y < A[x], \forall y \in A_{hi}, A[x] < y$, 这时 $A' = [A[l...x-1], A[x], A[x+1...r]]$ 为有序的.

1. 当 $k = 1$ 时.$A[1..1]$ 显然满足要求.

2. 当 $k = n$ 时. 设 $A[1...n]$ 已然有序, 则设 $A[n+1]$ 执行插入操作 Insert$(A[n+1], A[1...n])$ 后 $A[1...n]$ 分为 $A_l = A[1...x-1], A[x], A_r = A[x+1...n]$. 因为 A_l 和 A_r 内部已然有序,A_l 与 A_r 之间已然有序, 只需有 $\forall y \in A_l, y < A[x], \forall y \in A_r, A[x] < y$ 即可. 因此根据假设 $A[1...n+1]$ 是有序的.

3. 根据数学归纳法及 1,2, 对 $A[1...n](n \geqslant 1$ 进行增量 Insert 总是能够对一个数组排序.

1.2 算法复杂度

该排序为原地排序.

当我们朴素地按照定义实现 Insert 时. 它的复杂度是 $\Omega(1), \mathrm{O}(n)$ 的, 那么:

空间复杂度为 $\Theta(1)$, 时间复杂度为 $\Omega(n), \mathrm{O}(n^2)$.

1.2.1 最好情况举例

设 $A[1...n] = [1, 2, ..., n]$, 有每次 Insert 的复杂度为 $\Theta(1)$, 因此 $T(n) = n\Theta(1) = \Theta(n)$.

1.2.2 最差情况举例

设 $A[1...n] = [n, n-1, ..., 1]$, 有每次 Insert 的复杂度为 $\Theta(i)(1 \leqslant i \leqslant n)$, 因此:

$$T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2).$$

1.2.3 平均情况

设 $A[1...n]$ 为随机数据. 那么 Insert 的移动期望为 $O(\frac{i}{2}) = O(i)(1 \leqslant i \leqslant n)$. 从而估计的平均时间复杂度为:

$$T(n) = \sum_{i=1}^{n} \mathrm{O}(i) = \mathrm{O}(n^2).$$

2 朴素归并排序 MergeSort

分治 $A[1...\frac{n}{2}], A[\frac{n}{2}...n]$ 分治排序.$\Theta(n)$ 完成一次归并.

2.1 正确性

采用数学归纳法.

假设 (依然是数学归纳法, 此略):

定义 Merge$(A[1...n], B[1...n])$ 为归并操作, 它使得若 $A[1...n], B[1...n]$ 均为有序数组, 那么 $C[1...2n] =$ Merge$(A[1...n], B[1...n])$ 依然为有序数组.

1. 当 $k = 1$ 时.$A[1..1]$ 显然满足要求.

2. 设 $k < n$ 时归并排序 MergeSort$(A[1...k])$ 能够完成排序.

   当 $k = n$ 时. 做一次 MergeSort$(A_l = A[1...n/2])$, MergeSort$(A_r = A[n/2...n])$, 根据归纳假设, 这两个为有序数组. 做一次 Merge(A_l, A_r), 根据我们最初对 Merge 的假设,$A[1...n] =$ Merge(A_l, A_r) 为有序数组, 从而 MergeSort 能够完成规模为 n 的排序.

3. 根据数学归纳法及 1,2,MergeSort 能够完成规模为 $n(n \geqslant 1)$ 的排序.

2.2 算法复杂度

该排序为非原地排序.

当我们朴素地按照定义实现 Merge 时. 它的复杂度是 $\Theta(n)$ 的, 那么:

空间复杂度为 $\Theta(n)$, 时间复杂度为 $\Theta(n \log n)$.

2.2.1 最好情况和最差情况

该算法因为有复杂度为 $\Theta(n)$ 的 Merge, 所以它是时间复杂度稳定的.

2.2.2 平均情况

写出递归复杂度表达式:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n).$$

根据主定理 $T(n) = \Theta(n \log n)$.

3 混合插入归并排序 MixedMergeSort

定义如下:

$$\text{MixedMergeSort} := \begin{cases} \text{InsertionSort} & ,n < k; \\ \text{MergeSort} & ,n \geqslant k. \end{cases}$$

仅仅优化常数, 故不对其做复杂度分析.

4 多路归并排序 MultiwayMergeSort

分治

$$A[1...n] = [A[1...\frac{n}{k}], A[\frac{n}{k}...\frac{2n}{k}], ..., A[\frac{(k-1)n}{k}]...n],$$

再用 MultiMerge 在 $\Theta(n \log k)$ 的时间里合并.

多路归并排序正确的关键:

定义 MultiMerge$(A_1, A_2, ...A_m)$ 为归并操作, 它使得若 $A_1, A_2, ...A_m$ 均为有序数组, 那么 $C[1...\sum_{i=1}^{m} len[A_i]] =$ Merge$(A_1, A_2, ...A_m)$ 依然为有序数组.

本实验尝试了一个比较 toy 的实现方式, 而真正有价值的多路归并排序应当是将数据按块分发任务，再通过 burstI/O 和并发完成排序, 时间有限, 故从略.

5 快速排序 QuickSort

$\Theta(n)$ 完成一次划分 Partition. 再分治 $A[1...\text{pivot}(A)-1], A[\text{pivot}(A)+1...n]$ 分治排序. 朴素快速排序中 $\text{pivot}(A) = A[1]$.

5.1 算法复杂度

该排序为原地排序. 空间复杂度为 $\Theta(1)$, 时间复杂度为 $\Omega(n\log n), \mathrm{O}(n^2)$.

5.1.1 最好情况

假设选择的枢轴元素总是为 $A[1...n]$ 的中数 $A'[\frac{n}{2}]$. 那么有:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n\log n).$$

例如 $[3,2,1,4,5]$, 未曾尝试构造大数据中的最好情况.

5.1.2 最坏情况

假设选择的枢轴元素总是为 $A[1...n]$ 的最值元素 $A'[1]$ *or* $A'[n]$. 那么有:

$$T(n) = T(1) + T(n-1) + \Theta(n) = nT(1) + \sum_{i=1}^{n-1}\Theta(n) = \Theta(n^2).$$

例如:

1. $A[1...n] = [1,2,...n]$

2. $A[1...n] = [n,n-1,...1]$

3. $A[1...n] = [c,c,...c]$

5.1.3 平均情况

一般实现中 Partition 仍然不会将 $A[1...\text{pivot}(A)-1], A[\text{pivot}(A)+1...n]$ 有序化, 所以期望的复杂度为 $O(n\log n)$. 但在实验中, 发现了部分随机化数据能够让快速排序递归过深, 故快速排序仍需要像类似于 IntroSort 的优化.

5.2 Partition

根据划分的过程, 可以分为下面三种.

5.2.1 Hoare – Partition

Hoare – Partition 选择两侧逼近.

```
int hoare_partition (arr_element arr[], const int len)
{
    arr_element pivot = arr[0];
    for(int l = 0,r = len - 1;;) {
        while (l < r && arr[r] > pivot)r--;
        while (l < r && arr[l] <= pivot)l++;
        if (l >= r) {
            arr[0] = arr[l];
            arr[l] = pivot;
            return l;
        }
        std::swap(arr[l], arr[r]);
    }
    return -1;
}
```

5.2.2 Lomuto – Partition

Lomuto – Partition 选择单侧逼近.

```
int lomuto_partition (arr_element arr[], const int len)
{
    arr_element pivot = arr[0];
    int l = 0;
    for(int i = 1; i < len; i++) {
        if (arr[i] <= pivot) {
            l++;
            std::swap(arr[l], arr[i]);
        }
    }
    std::swap(arr[0], arr[l]);
    return l;
}
```

5.2.3 Median3 – Partition

Median3 – Partition 选择先选某三个数中的中位数, 然后再进行划分.

```
int hoare_partition_with_median_of_three (arr_element arr[], const int len)
{
    int m = len >> 1;
    if (arr[len – 1] < arr[0]) {
        std::swap(arr[len – 1], arr[0]);
    }
    if (arr[len – 1] < arr[m]) {
        std::swap(arr[len – 1], arr[m]);
    }
    if (arr[m] < arr[0]) {
        std::swap(arr[m], arr[0]);
    }

    return hoare_partition(arr, len);
}
```

5.2.4 其他 Partition

1. Randomized – Partition 选择一个 random 的 pivot 进行划分.

2. 两侧夹逼划分, 防止因为同一元素个数过多而退化.

6 随机快速排序 RandomQuickSort

先在程序入口做一次 RandomShuffle, 再进行快速排序.

6.1 算法复杂度

6.1.1 最好情况

假设选择的枢轴元素总是为 $A[1...n]$ 的中数 $A'[\frac{n}{2}]$. 那么有:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n).$$

例如 $[3, 2, 1, 4, 5]$, 未曾尝试构造大数据中的最好情况.

6.1.2 最坏情况

假设选择的枢轴元素总是为 $A[1...n]$ 的最值元素 $A'[1]$ *or* $A'[n]$. 那么有:

$$T(n) = T(1) + T(n-1) + \Theta(n) = nT(1) + \sum_{i=1}^{n-1} \Theta(n) = \Theta(n^2).$$

例如:

1. $A[1...n] = [c, c, ...c]$

6.1.3 平均情况

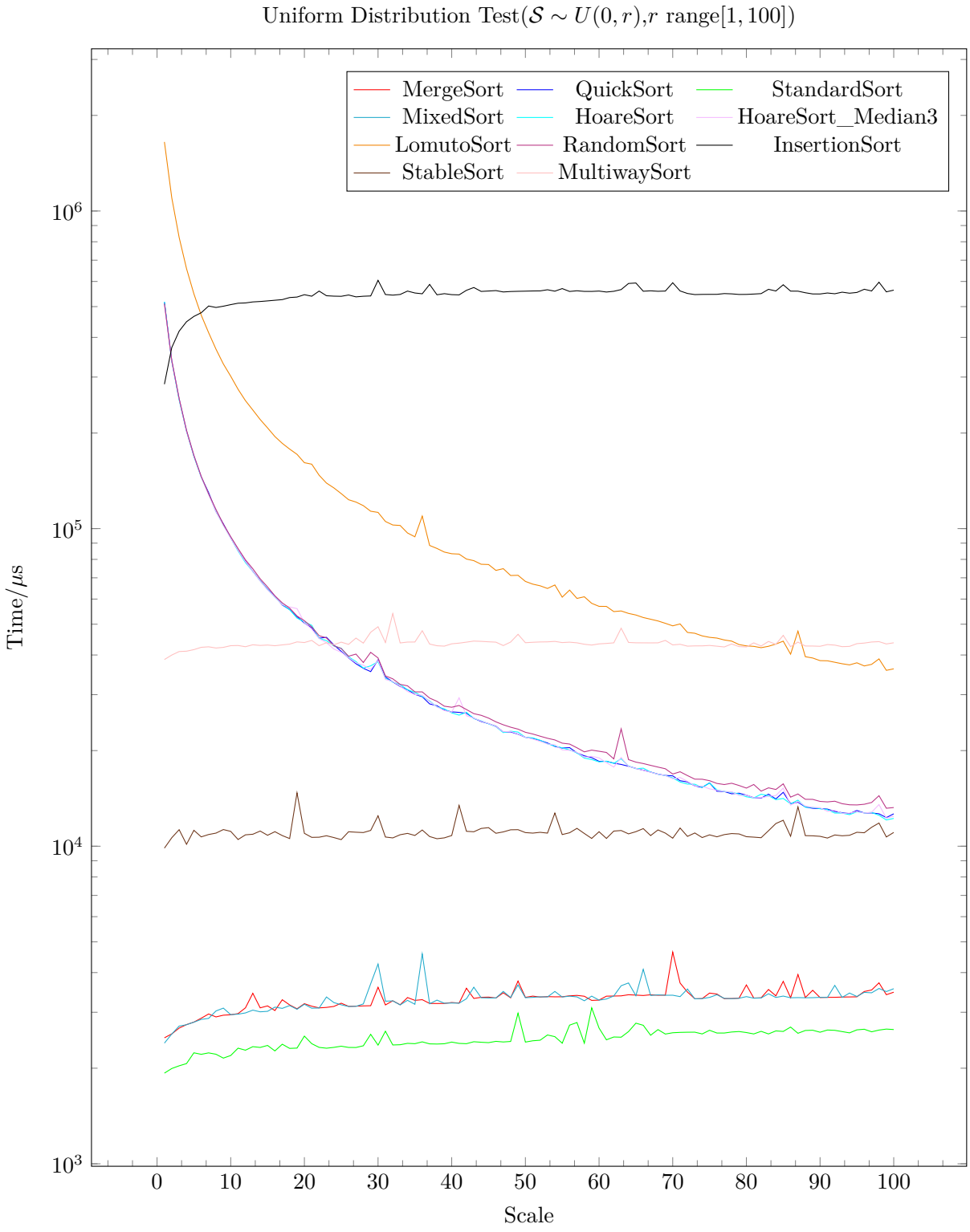
一般实现中 Partition 仍然不会将 $A[1...\text{pivot}(A) - 1], A[\text{pivot}(A) + 1...n]$ 有序化, 所以期望的复杂度为 $O(n \log n)$. 但在实验中, 发现了部分随机化数据能够让快速排序递归过深 (同学测试每次都随机化仍有可能被卡).

7 稳定快速排序 StableQuickSort

先在程序入口做一次 RandomShuffle, 再进行快速排序. 增加一个 Position 作为第二 key, 并对 $Pair[1...n] = [\text{MakePair}(A[i], i)...]$ 排序, 且 $pivot(Pair)$ 为所有具有 key 为 $A[1]$ 的中间值.

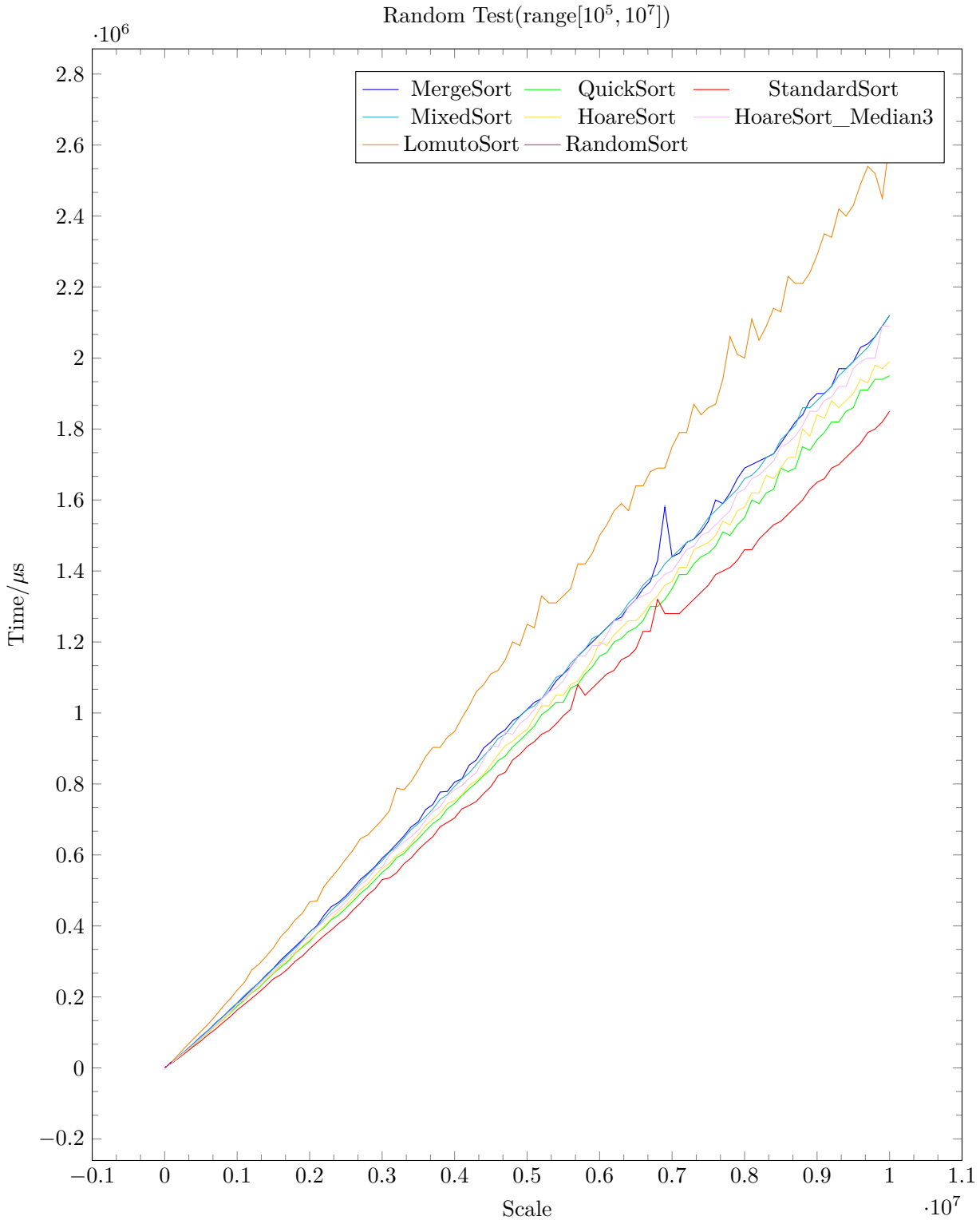
8 数据测试 (Uniform Distribution Test)

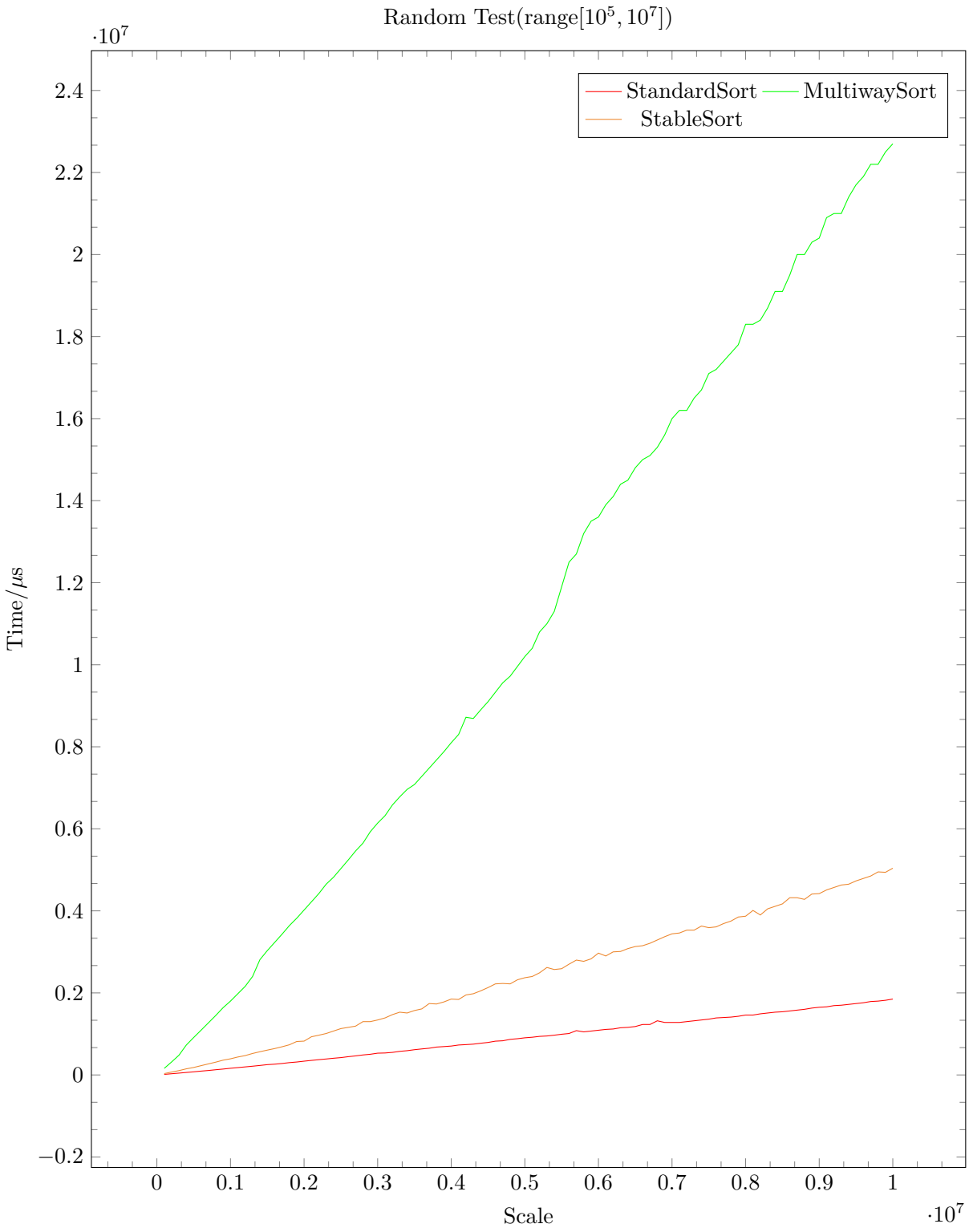
使用 uniform_distribution_testset 数据集.generate.h 还有其他分布的生成函数, 时间有限, 故略.



9 数据测试 (Random Test)

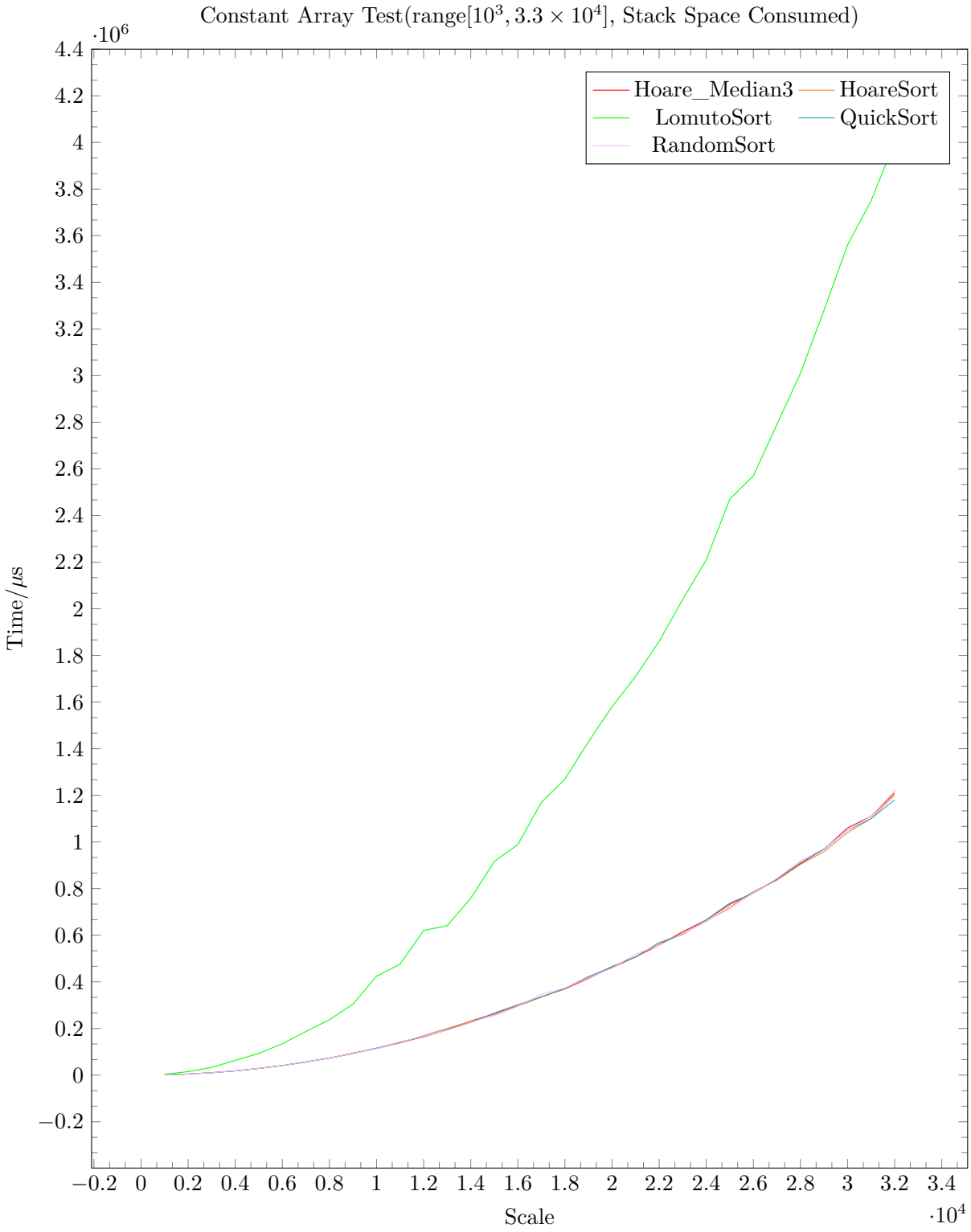
使用 random_testset 数据集.InsertionSort 因为复杂度太高, 无法测试.



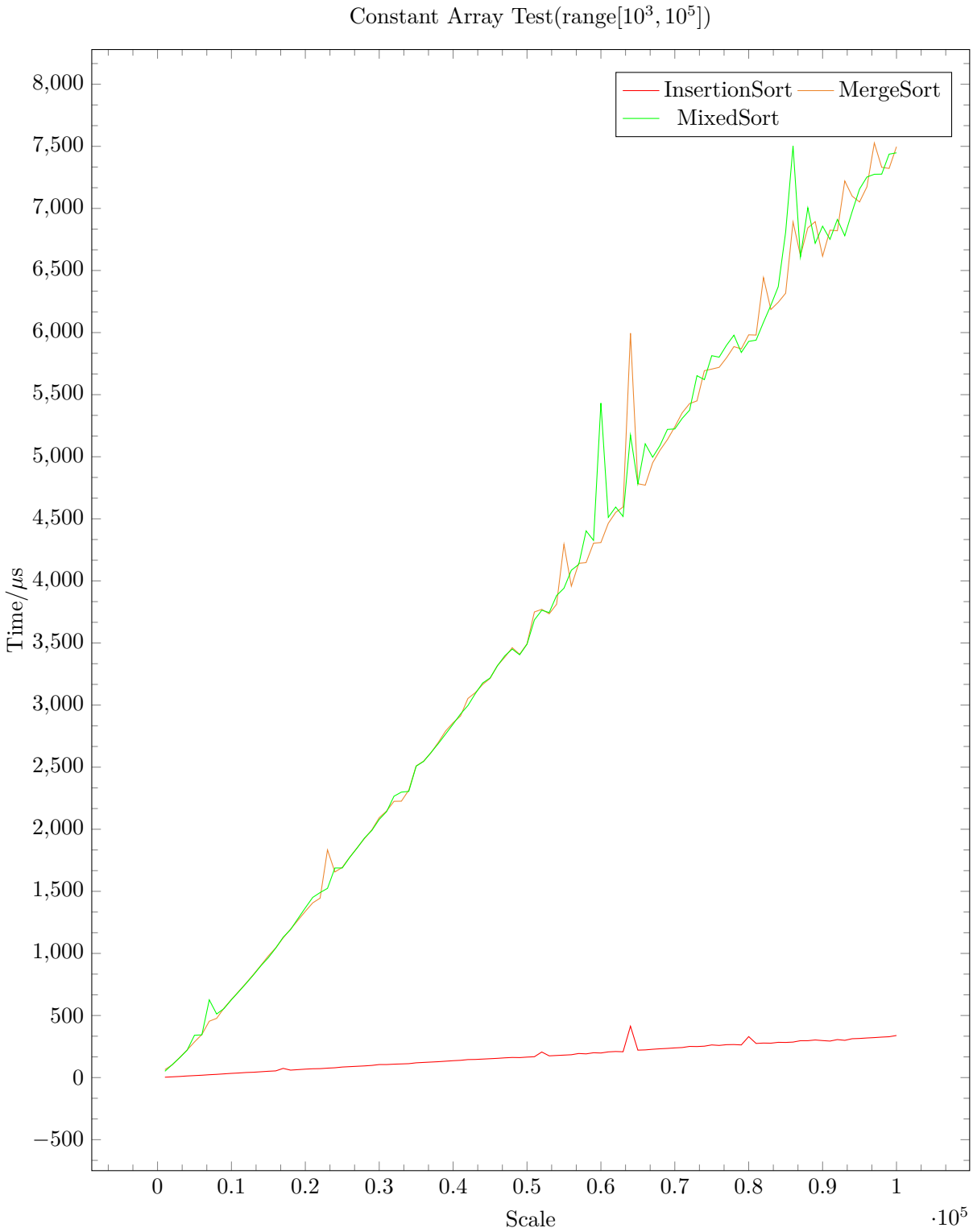

Random Test(range[$10^5, 10^7$])
$\cdot 10^7$
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
Time/μs
StandardSort
MultiwaySort
StableSort
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Scale
$\cdot 10^7$

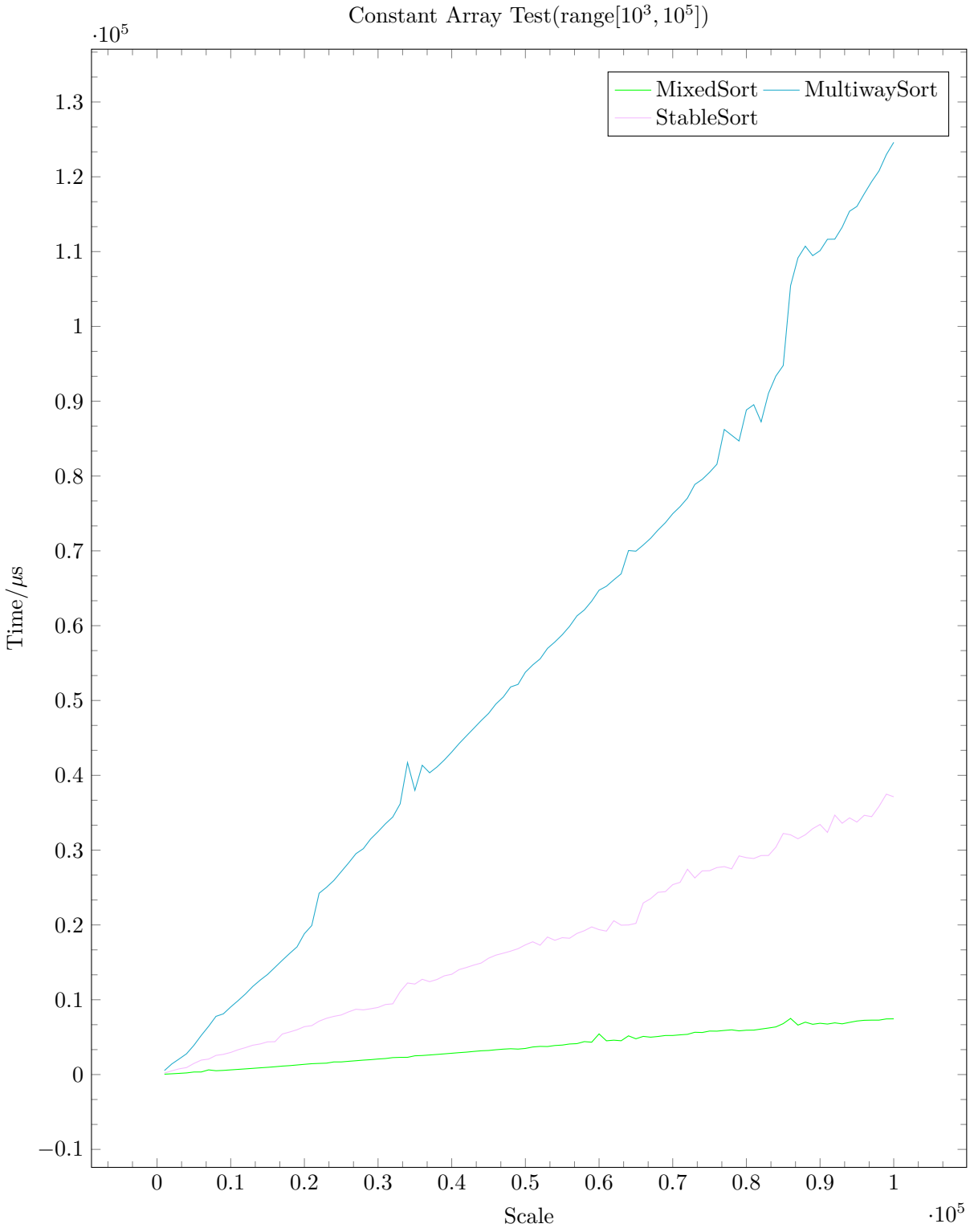
10 数据测试 (Constant Array Test)

使用 constant_testset 数据集, 下图为发生栈溢出的排序.



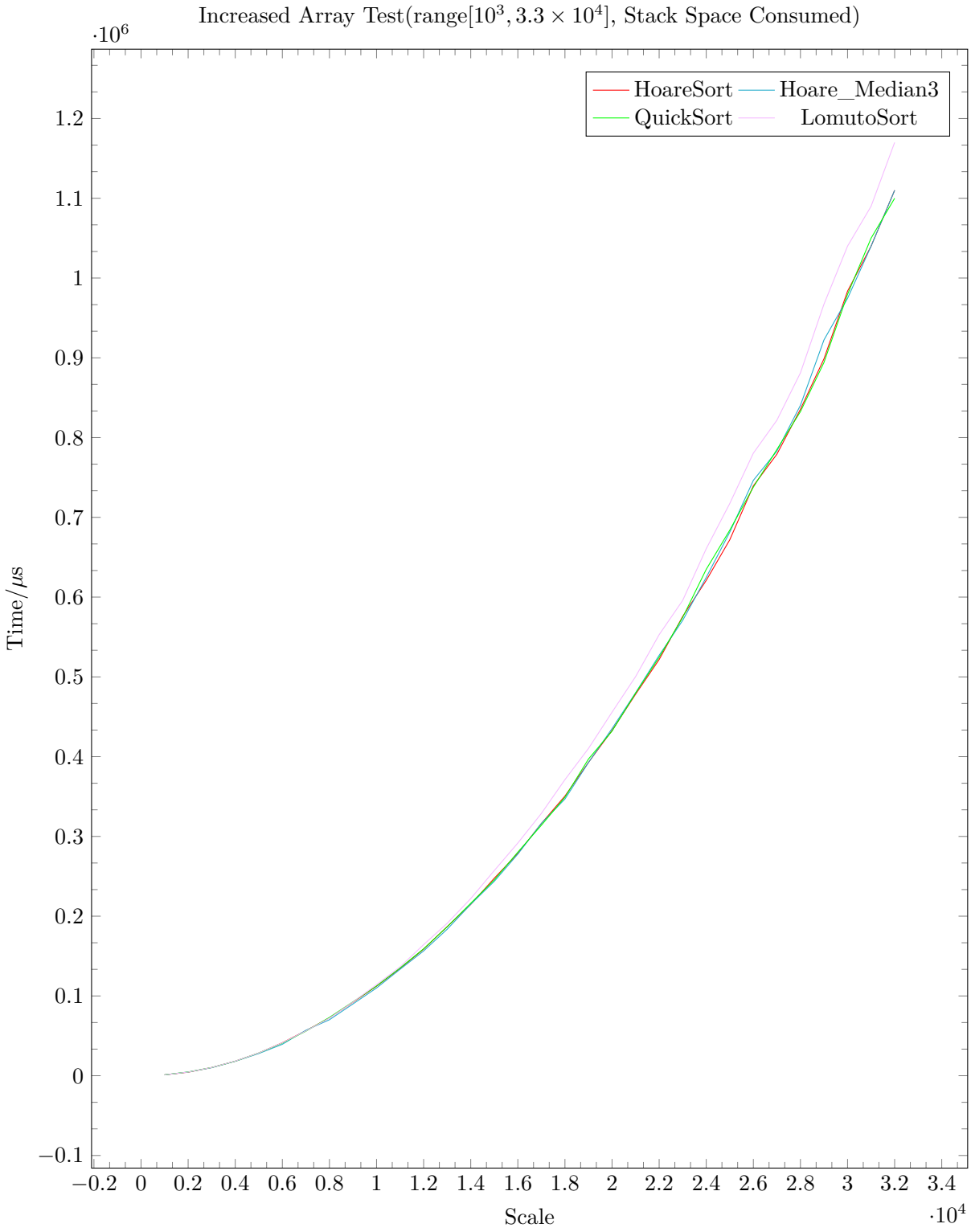
下两图为正常完成测试的排序.



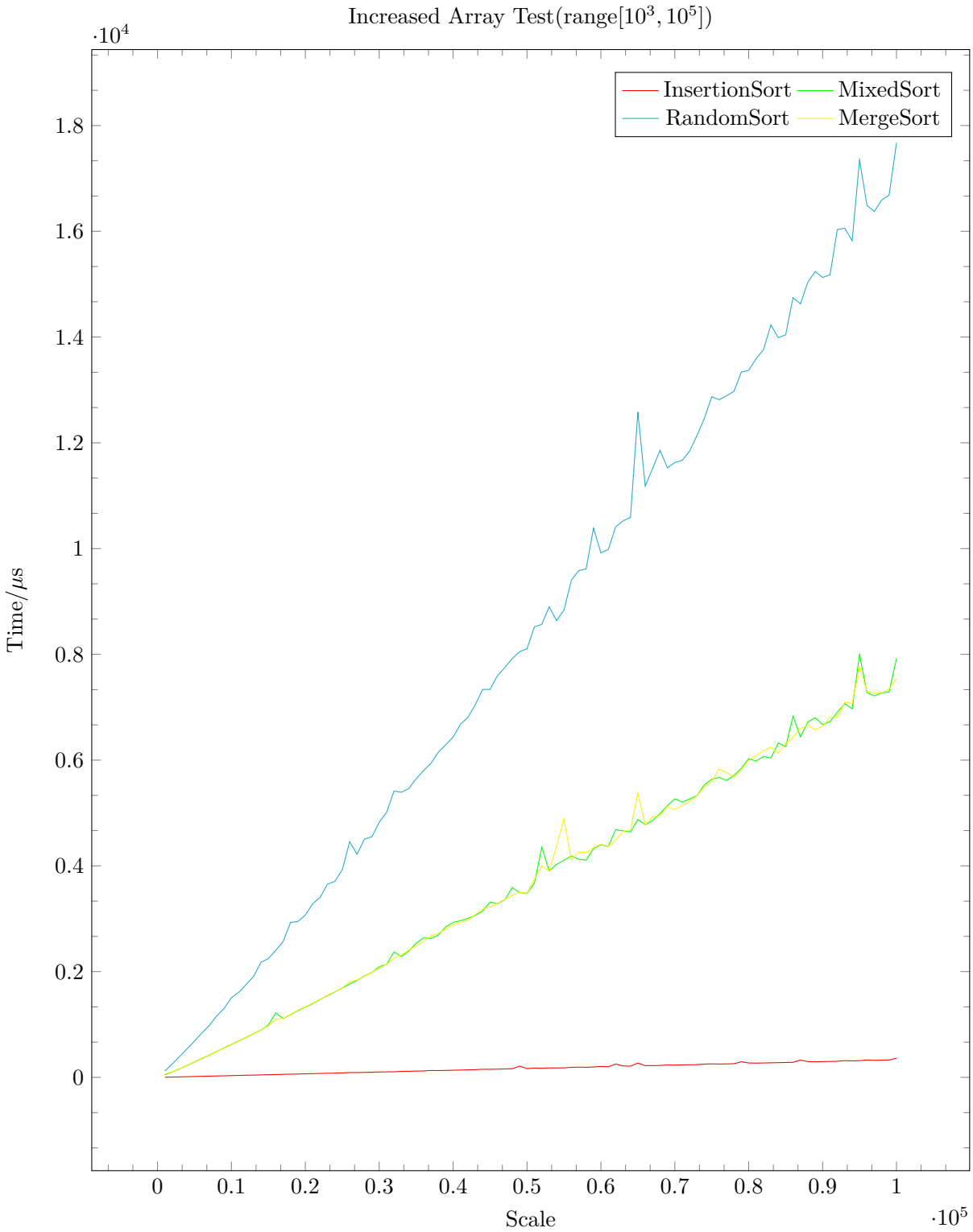

Constant Array Test(range[10^3, 10^5])
$\cdot 10^5$
MixedSort
MultiwaySort
StableSort
Time/μs
Scale
$\cdot 10^5$
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
−0.1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

11 数据测试 (Increased Array Test)

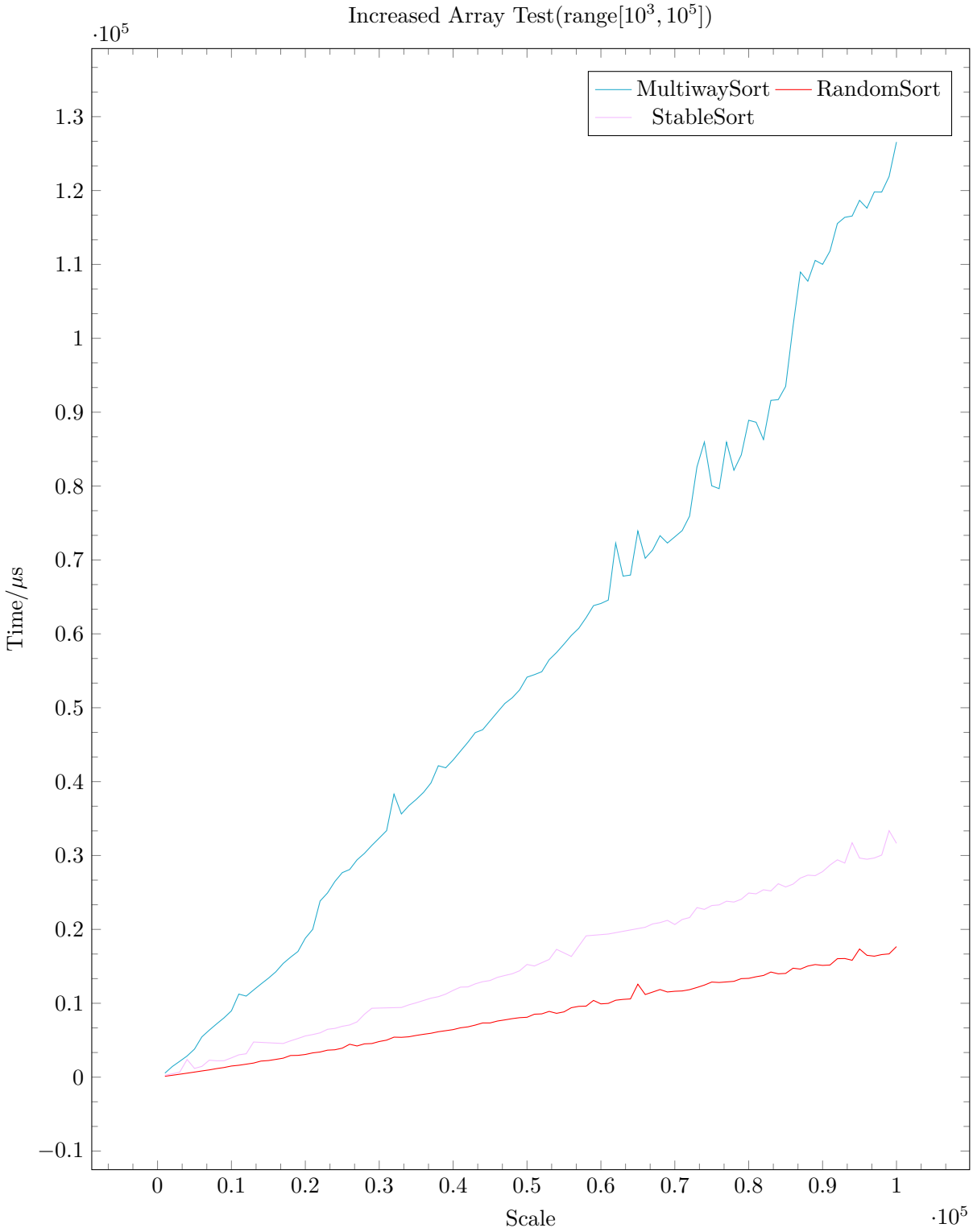
使用 increase_testset 数据集, 下图为发生栈溢出的排序.



下两图为正常完成测试的排序.

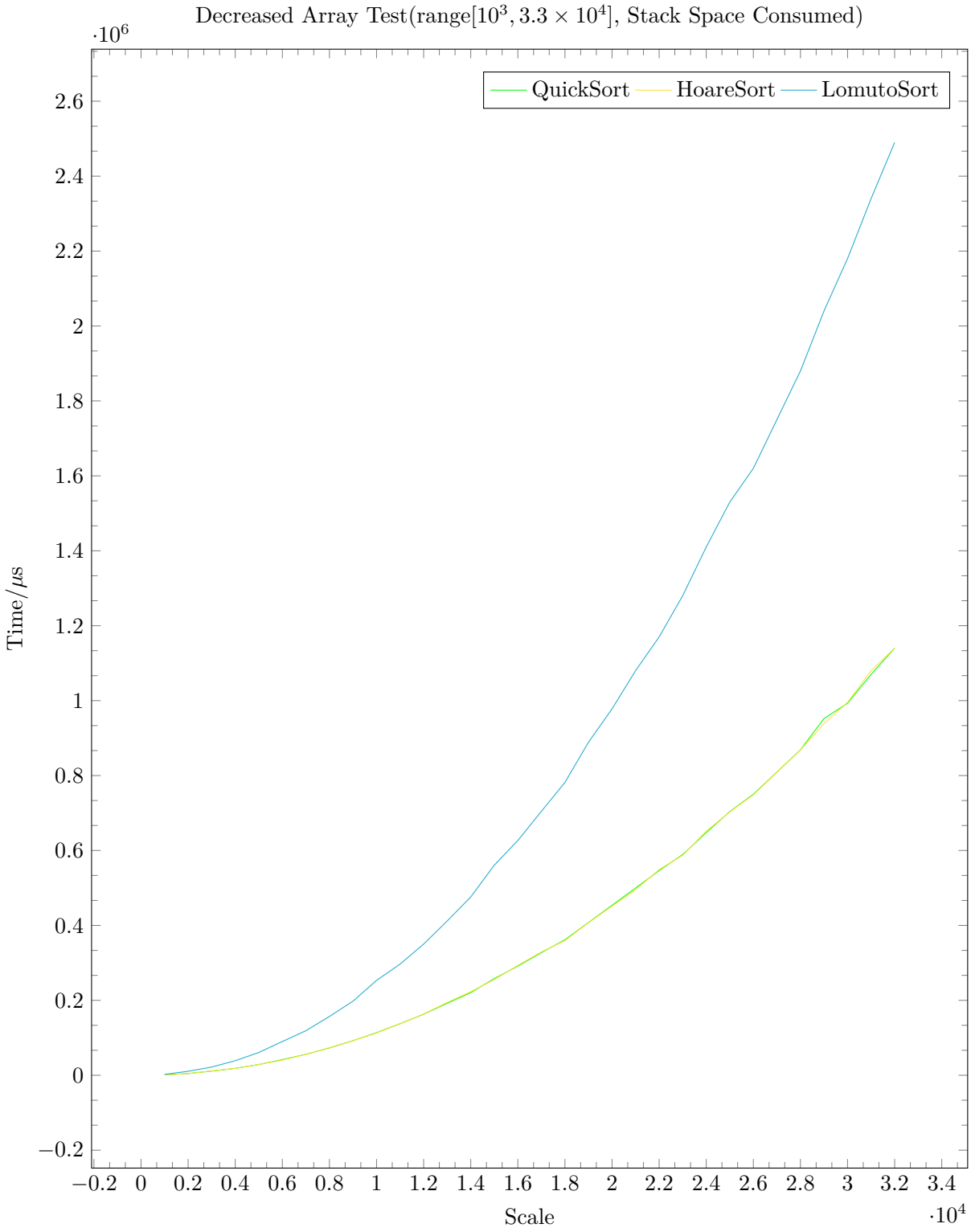


在 StableSort 测试的过程中, 可能有的测试点发生了 CPU 降频或 CPU 中断, 故在.dat 中去除了部分测试点.

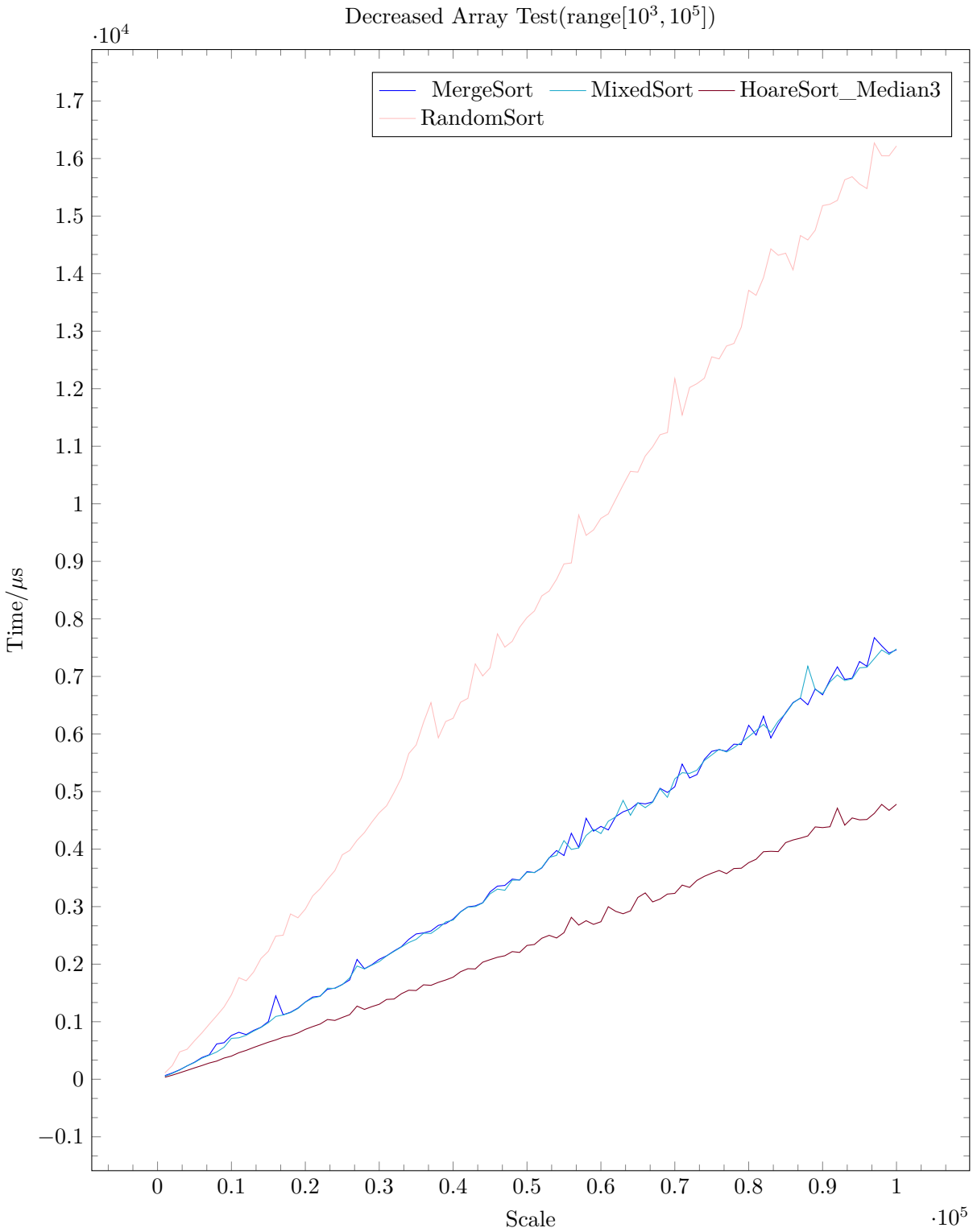


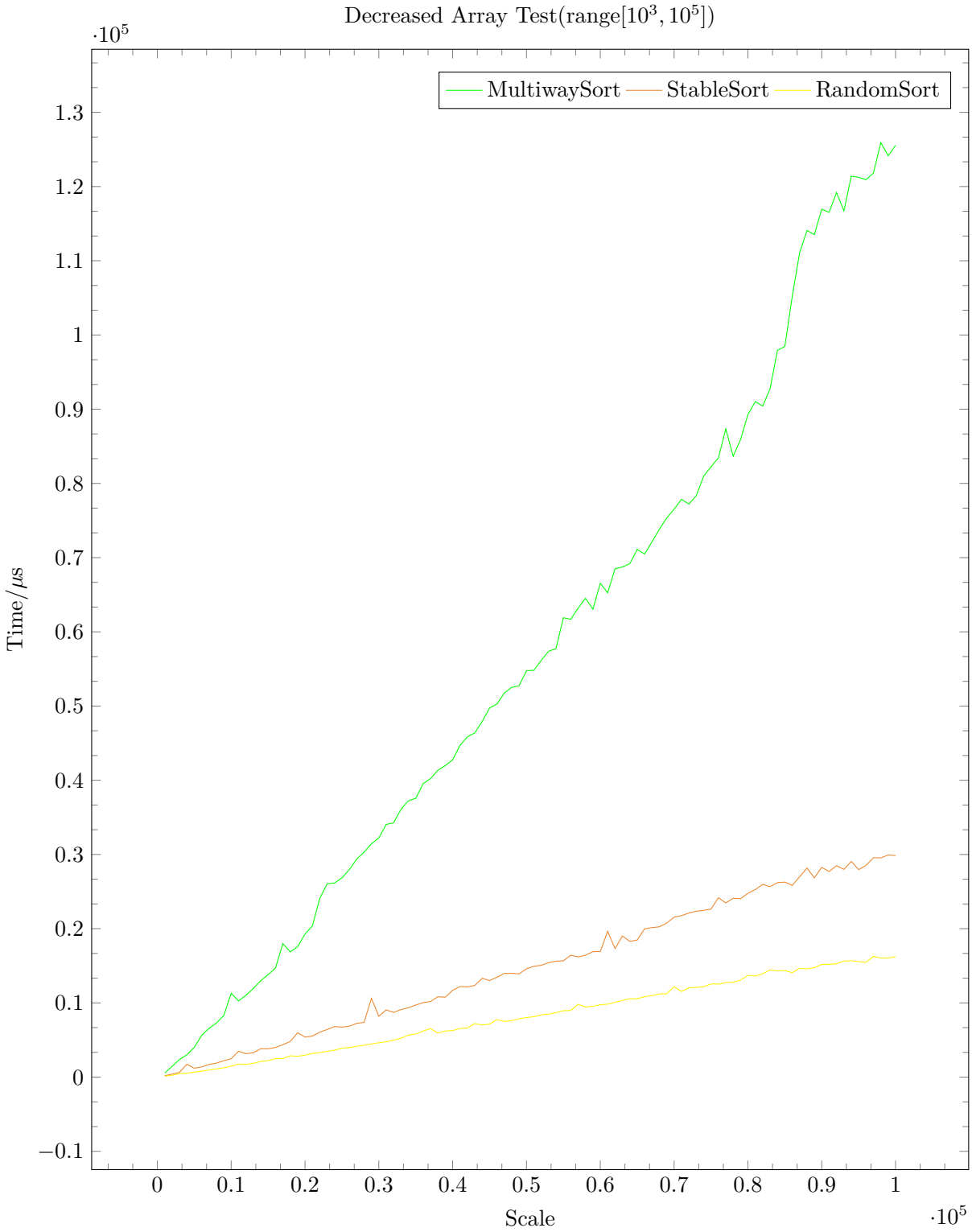
12 数据测试 (Decreased Array Test)

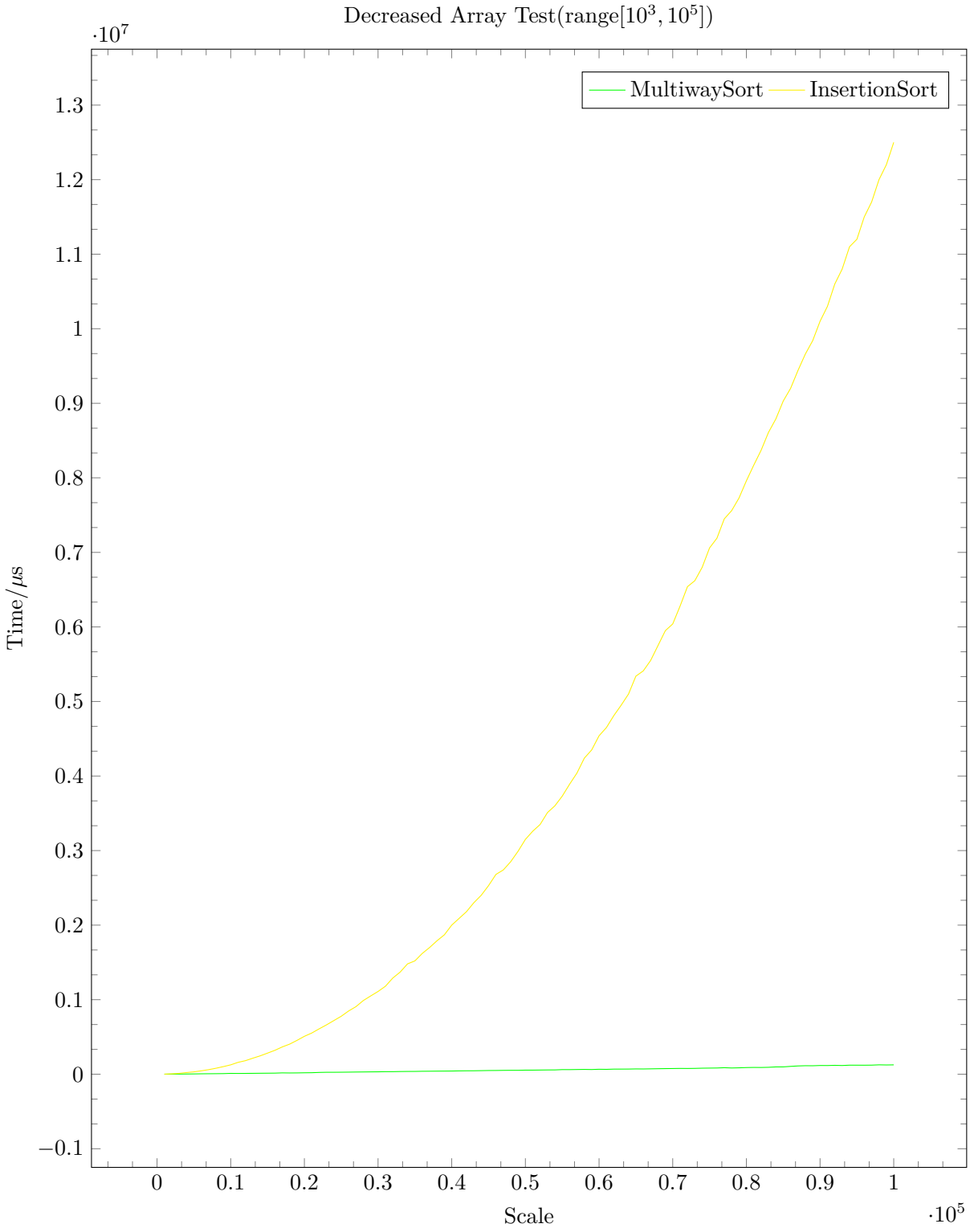
使用 decrease_testset 数据集, 下图为发生栈溢出的排序.



下三图为正常完成测试的排序.




Decreased Array Test(range[$10^3, 10^5$])
$\cdot 10^5$
MultiwaySort
StableSort
RandomSort
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
−0.1
Time/μs
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Scale
$\cdot 10^5$


Decreased Array Test(range[$10^3, 10^5$])
$\cdot 10^7$
MultiwaySort
InsertionSort
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
−0.1
Time/μs
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Scale
$\cdot 10^5$

13 总结

根据 Uniform Distribution Test.MergeSort 和 MixedMergeSort 对数组的有序度不敏感. 仅仅比标准库排序高了一点常数.

其次是 StableQuickSort, 但 StableQuickSort 牺牲了空间, 让相较于归并排序的优势也被损失.

再往上是 MultiwayMergeSort. 因为这是内排序中的比较,IO 不是主导的时间因素, 且本实验没有将 MultiwayMergeSort 良好实现. 故下文也不再对其分析.

本实验唯一的 $O(n^2)$ 算法 InsertionSort 自然排在最上面.InsertionSort 左侧有短暂的上升段, 因为 InsertionSort 在有序数列中大量的节省了腾挪次数, 从而做到 "$O(n)$" 的排序.

除了这些对有序度不敏感的排序还有对其敏感的一系列快速排序.LomutoSort 拥有最差的表现, 因为它的划分比其他的排序要慢得多 (表现在单向的累积 Swap 上, 其他基于 Hoare 的 Partition 是同时 Swap 高低数的, 所以几乎减少了一倍多的时间).

在 Random Test 中, 数据是随机的, 因而一系列快速排序均接近标准排序. 而 StableQuickSort 在随机化数据上不占优势的弱点也体现出来了.

在 Constant Array Test 中, 数据是完全常值的, 因而 RandomSort 的伎俩没有奏效,Median3 的伎俩也没有奏效. 其他几个自然也无法抵御. 从而 6 个快速排序只有 StableQuickSort 抵御了常值数列的攻击.

在 Increased Array Test 中, 数据是递增的, 值得注意的是 Median3 居然没有抵御成功, 原因应该是实现问题 (不确定).

在 Decreased Array Test 中, 数据是递减的, 这一轮中只有三个朴素的快速排序败倒.

还有一个值得注意的现象是 MixedMergeSort 和 MergeSort 几乎是同样的排序, 可能需要进行参数调整.

在大数据的测试上.CPU 和内存都没能跑满, 而 I/O 成为了主要限制, 故笔者寻找了多路归并排序, 但时间有限并未测试.

除此还有笔者还有一个意外发现, 那就是一个随机数据集 (规模为 3×10^6) 竟让除了 StableSort 以外其他所有的快速排序都消耗完了栈空间, 经同学测试他实现的 RandomizedSort(本文快速排序中介绍的第二个基于 RandomPartition 的排序算法) 也被卡住了. 看来随机化快速排序还是不能大大消除有序化对快速排序

的影响, 必须另寻它路 (比如文中提到的 IntroSort). 该数据集并未删除, 可以在源码中找到.

时间有限, 还有很多测试没做 (比如其他一些分布的数据集测试, 比如大数据下的排序算法), 排序也是朴素而普通地完成了. 如果有时间, 可能会再做大数据 I/O. 作者实现了比较简单的 Integer 数据集测试框架, 如果有人有兴趣也可以自己生成数据集和编写排序链接到框架上进行测试.