随机算法课程

实验报告

实验二：QuickSort 算法的再探讨

姓名：袁野

学号：1190200122

班级：1903102

评分表：（由老师填写）

最终得分：	
对实验题目的理解是否透彻：	
实验步骤是否完整、可信 ：	
代码质量 ：	
实验报告是否规范 ：	
趣味性、难度加分 ：	
特 色：	1
	2
	3

一、实验题目概述

QuickSort 是一种特别基础常见的排序算法，在该实验中，通过复现 QuickSort 的一种实现方式，探究该种实现方式在对不同的元素重复程度的数据排序时的现象以及原因，同时针对该现象进行改进，编写新的程序进行探究，同时与标准库中 QuickSort 算法进行对比，从而进一步认识理解随机算法复杂性分析结果的有效性。

二、对实验步骤的详细阐述

1、严格按照实验指导书上的伪代码实现一种比较基础的 QuickSort 算法。

2、按照实验指导书的要求，生成 11 个排列均匀的数据文件。

3、运行复现的 QuickSort 代码，会发现在 Ubuntu 环境下只有完全不同数字的数据可以迅速跑出答案，而 10%重复元素的数据跑了 43s 之后返回结果，但是第三个数据跑了一段时间后发生段错误从而终止运行。

4、对于该现象的原因我们将在第四部分进行分析，而针对原因对代码进行改进并运行，我们发现该代码对于 11 个数据文件全都迅速跑出结果且结果正确。改进后的代码如下：

```
int Rand_Partition(int *A, int p, int r) {
    int i = rnd.next(p, r);
    int x = A[i];
    swap(A[i], A[r]);
    int first = p;
    int last = r-1;
    while (1) {
        while (A[first] < x) first++;
        while (A[last] > x) last--;
        if (first >= last) {
            swap(A[first], A[r]);
            return first;
        }
        swap(A[first], A[last]);
        first++;
        last--;
    }
}
```

5、通过调用 STL 中的 sort 函数，和改进的 QuickSort 算法进行对比。

6、我将三种不同的方式定义在了 main.cpp 的不同命名空间中，而且为了保证最终排序结果的正确性，我们在排序之后进行了判定和时间的记录。

```
int clk = clock();
if (o == 1) solve1::QuickSort(A, 0, n-1);
else if (o == 2) solve2::QuickSort(A, 0, n-1);
else if (o == 3) solve3::QuickSort(A, 0, n-1);
else cerr << "Error ! \n"
clk = clock()-clk;
for (int i=1; i<n; ++i) assert(A[i] >= A[i-1]);
cerr << "Test " << t << " OK ! \n";
outFile << "Test data " << t << " Time: " << 1.0*clk/CLOCKS_PER_SEC << endl;
inFile.close();
```

三、实验数据

1. 实验设置

实验环境：

Ubuntu20.04.4 LTS(GNU/Linux 5.10.102.1-microsoft-standard-WSL2 x86_64)

数据：

生成思路为：首先构造一个$A_i = i$的序列，然后将前固定长度的位置全部变为同一个随机数P，然后将其打乱为随机序列。由于 C++自带的随机函数生成器

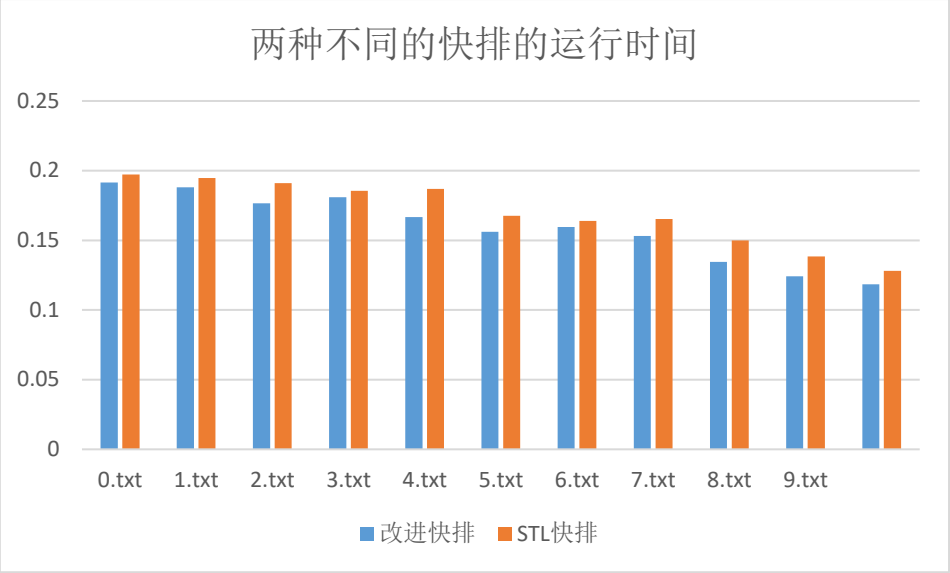
存在种种局限性，因此我采用的是 testlib.h 库(地址)中的随机数生成器构造数据文件。

数据集名称	重复数据占比	数据集大小
0.txt	0%	10^6
1.txt	10%	10^6
2.txt	20%	10^6
3.txt	30%	10^6
4.txt	40%	10^6
5.txt	50%	10^6
6.txt	60%	10^6
7.txt	70%	10^6
8.txt	80%	10^6
9.txt	90%	10^6
10.txt	100%	10^6

2. 实验结果

原始版本的 QuickSort 仅能在前两个数据集上运行成功，改进后和 STL 均可以在全部的数据集上运行成功，运行时间如下：

数据集	原版代码	改进代码	STL：sort
0.txt	0.243521s	0.19148s	0.19712s
1.txt	43.6652s	0.188045s	0.19474s
2.txt	-	0.176469s	0.191017s
3.txt	-	0.18089s	0.185573s
4.txt	-	0.166657s	0.186981s
5.txt	-	0.1561s	0.167657s
6.txt	-	0.159483s	0.163874s
7.txt	-	0.153225s	0.165293s
8.txt	-	0.134609s	0.149938s
9.txt	-	0.124263s	0.13843s
10.txt	-	0.118391s	0.128063s



四、对实验结果的理解和分析

1、最原始的 QuickSort 中导致栈溢出是在于其划分策略，它会从左往右依次检查每一个值与基准值的关系，并且以最终停下的位置作为其划分的位置，这样的策略在元素重复率足够低的数据集上的期望效果是良好的，但是如果遇到了元素完全一样的数据集，他会依次从左往右扫描所有的数据集并最终停在最右端，以最右端作为递归下去的划分位置，即将原先有n个元素的序列划分为一个$n-1$长度的和1长度的序列，这样会导致排序的期望复杂度由原先的 $O(nlogn)$ 退化为$O(n^2)$，导致效率严重降低，同时，由于划分效果差，导致递归深度大大增加，原先 $O(logn)$ 的递归深度会退化为 $O(n)$ ，这会在计算机运行时造成栈溢出，从而导致段错误，经测试，Windows 环境下只能成功运行元素完全不同的数据，其他的数据，10%重复元素的数据就会发生栈溢出，而 Ubuntu 前两个数据可以成功运行，20%重复元素的数据就会发生栈溢出。

2、改进之后的 QuickSort 修改了划分方式，从当前数据序列的两端开始扫描，如果遇到左侧数字大于标准基值且右侧数字小于标准基值后，会将二者进行交换，最终当两个指针相遇之后返回其中一个指针的位置即可。这样的好处是即使面对元素完全一样的区间，也可以两端同步向中间移动，从而最终停在序列的正中间，从而产生一个标准的二分，这种策略下的划分方式都是良好的，最终的结果也显示，这样的程序可以在所有的数据上成功运行，且时间效率与 STL 近似甚至略优于 STL。

五、实验过程中最值得说起的几个方面

通过查找网络资料和阅读 algorithm 库中 sort 部分的代码我们可以发现其实 STL 的 sort 函数并不是简单的快速排序算法，它在快速排序的基础上有如下的改进：

1、 其实 sort 的源代码中快速排序被替换成了 Introspective Sorting(内省式排序)，它是一种混合式的排序算法，集成了快速排序，堆排序，插入排序的优点。在数据量很大时采用正常的快速排序，此时效率为O(logN)；一旦分段后的数据量小于某个阈值，就改用插入排序，因为此时这个分段是基本有序的，这时效率可达O(N)；在递归过程中，如果递归层次过深，分割行

为有恶化倾向时，它能够自动侦测出来，使用堆排序来处理，在此情况下，使其效率维持在堆排序的O(N logN)，但这又比一开始使用堆排序好。

```
template<typename _RandomAccessIterator, typename _Size, typename _Compare>
  void
  __introsort_loop(_RandomAccessIterator __first,
                   _RandomAccessIterator __last,
                   _Size __depth_limit, _Compare __comp)
  {
    while (__last - __first > int(_S_threshold))
    {
      if (__depth_limit == 0)
        {
          std::__partial_sort(__first, __last, __last, __comp);
          return;
        }
      --__depth_limit;
      _RandomAccessIterator __cut =
        std::__unguarded_partition_pivot(__first, __last, __comp);
      std::__introsort_loop(__cut, __last, __depth_limit, __comp);
      __last = __cut;
    }
  }
```

上图即为从库函数中截取出来的 sort 函数的主题部分，可以发现，函数使用了_S_threshold 常量来循环控制分段递归操作，使用__depth_limit 变量来控制递归深度，当超出该阈值时采用堆排序，即 std::__partial_sort 函数，从而保证其不会发生栈溢出现象。

2、 而对于标准值的选取方式，标准库函数也并没有简单采用随机序列值的方式，而是采用了三点中值法，其函数实现如下：

```
/// This is a helper function...
template<typename _RandomAccessIterator, typename _Compare>
  inline _RandomAccessIterator
  __unguarded_partition_pivot(_RandomAccessIterator __first,
                              _RandomAccessIterator __last, _Compare __comp)
  {
    _RandomAccessIterator __mid = __first + (__last - __first) / 2;
    std::__move_median_to_first(__first, __first + 1, __mid, __last - 1,
                                __comp);
    return std::__unguarded_partition(__first + 1, __last, __first, __comp);
  }
```

它的作用是取首部、尾部和中部三个元素的中值作为 pivot。我们之前学到的快速排序都是选择首部、尾部或者中间位置的元素作为 pivot，并不会比较它们的值，在很多情况下这将引起递归的恶化。现在这里采用的中值法可以在绝大部分情形下优于原来的选择。

3、 当__introsort_loop 函数进行完之后便会回到 sort 函数的主体，如下：

```
template<typename _RandomAccessIterator, typename _Compare>
  inline void
  __sort(_RandomAccessIterator __first, _RandomAccessIterator __last,
         _Compare __comp)
  {
    if (__first != __last)
    {
      std::__introsort_loop(__first, __last,
                            std::__lg(__last - __first) * 2,
                            __comp);
      std::__final_insertion_sort(__first, __last, __comp);
    }
  }
```

此时分段递归排序的过程已经完成，而通过__introsort_loop 的源代码我们知道，此时__last 和__first 之间的距离，即剩余未排序的序列长度是不超过_S_threshold 这个常量的。这个常量的定义如下：

```
/**
 *  @doctodo
 *  This controls some aspect of the sort routines.
*/
enum { _S_threshold = 16 };
```

紧接着，函数会进入__final_insertion_sort 函数，这便是当区间小于阈值后进行插入排序的部分：

```
/// This is a helper function for the sort routine.
template<typename _RandomAccessIterator, typename _Compare>
  void
  __final_insertion_sort(_RandomAccessIterator __first,
                         _RandomAccessIterator __last, _Compare __comp)
  {
    if (__last - __first > int(_S_threshold))
    {
      std::__insertion_sort(__first, __first + int(_S_threshold), __comp);
      std::__unguarded_insertion_sort(__first + int(_S_threshold), __last,
                                      __comp);
    }
    else
  std::__insertion_sort(__first, __last, __comp);
  }
```

我们发现，即使是插入排序，设计者们也将其分成了__insertion_sort 和__unguarded_insertion_sort 两种不同的实现形式，实际上这两种实现形式的区别就在于前者有边界检查而后者没有边界检查，所以后者效率更高，但是对于__unguarded_insertion_sort 函数来说，一定得确保这个区间的左边有效范围内已经有了最小值，否则没有越界检查将可能带来非常严重的后果。

那么__final_insertion_sort 函数为什么要这样设计呢，通过多次简单的模拟快排过程我们不难发现，无论经过几次递归调用，对于所有划分的区域，左边区间所有的数据一定比右边小，这就保证了在插入排序的环境中，最小值存在于前 16 个元素之中，所以对于前十六个元素采用__insertion_sort，而对于后边的序列采用__unguarded_insertion_sort 可以保证不会越界出错，同时省去了多次边界判断产生的代价。

4、 以下函数即为标准库函数的快排主体部分:

```
/// This is a helper function...
template<typename _RandomAccessIterator, typename _Compare>
  _RandomAccessIterator
  __unguarded_partition(_RandomAccessIterator __first,
                        _RandomAccessIterator __last,
                        _RandomAccessIterator __pivot, _Compare __comp)
  {
    while (true)
    {
      while (__comp(__first, __pivot))
        ++__first;
      --__last;
      while (__comp(__pivot, __last))
        --__last;
      if (!(__first < __last))
        return __first;
      std::iter_swap(__first, __last);
      ++__first;
    }
    }
```

因此，我们发现，虽然标准库函数中的 sort 代码十分简短，但是其包含的细节技巧十足，在每一个细枝末节的地方追求效率的优化，使得其整体效率出众。上述对于 std::sort 源代码的探究参考 https://feihu.me/blog/2014/sgi-std-sort/#%E4%B8%BA%E4%BD%95__final_insertion_sort%E5%A6%82%E6%AD%A4%E5%AE%9E%E7%8E%B0

此外，为了正确运行测试代码，编写了 makefile 文件整合命令，具体可见 README。