学号：19S103256

姓名：文荟俨

实验四：快速排序

1. 实验目的

(1)掌握快速排序随机算法的设计思想与方法
(2)熟练使用高级编程语言实现不同的快速排序算法
(3)利用实验测试给出不同快速排序算法的性能以理解其优缺点

2. 实验问题

快速排序是算法导论中的经典算法。在本实验中，给定一个长为 n 的整数数组，要求将数组升序排序。

3. 实验步骤

(1) 随机快排
算法 **Random quicksort**(S)
输入 :无序数组 S
输出 :有序数组 S

```
1:  QuickSort(A, p, r)
2:    If p<r   then
3:      q = Rand_Partition(A, p, r)
4:      QuickSort(A, p, q-1)
5:      QuickSort(A, q+1, r)
6:  Rand_Partition(A, p, r)
7:    i=Random(p, r)
8:    swap(A[r], A[i])
9:    x=A[r]
10:   i=p-1
11:   for j=p to r-1
12:     If  A[j]<=x   then
13:       i=i+1, swap(A[i], A[j])
14:   swap(A[i+1], A[r])
15:   return i+1
```

该部分的核心代码如清单 1 所示。

清单 1 随机快排核心代码

```
void swap_int(int *a, int *b) {
      if (*a != *b) {
            int temp;
            temp = *a;
            *a = *b;
            *b = temp;
```

```
	}
}

int rand_partition(int *A, int p, int r) {
	srand((unsigned)time(0));
	int i = my_generate_node(p, r+1);
	swap_int(&A[r], &A[i]);
	int x = A[r];
	i = p - 1;
	for (int j = p; j < r; j++) {
		if (A[j] <= x) {
			i++;
			swap_int(&A[i], &A[j]);
		}
	}
	swap_int(&A[i + 1], &A[r]);
	return i + 1;
}
```

(2) 三路快排

由于随机快排在重复元素较多时递归树划分不均衡，排序速度慢，且存在爆栈的情况。本实验采用了四处改进以优化：(1)通过 Visual Studio 调整系统默认栈的大小，解决爆栈问题；(2)对于重复元素，不予交换，节省排序时间；(3)将原有的>v 和<v 的划分，增加为<v、>v 和=v；(4)采用双指针和 v 比较大小，节省排序时间。上述优化的示意图如图 1 所示。

图 1 三路快排优化示意图

改进后的算法如下:

算法 **Trple sort**(S)
输入 :无序数组 S
输出 :有序数组 S

1: 设 p 为 S 的最小下标，r 为 S 的最大下标
2: 从[p, r]中随机选一个元素作为划分标准 v
3: **If** 当前 i 指向的元素==v **then** i←i+1
4: **If** 当前 i 指向的元素<v **then** swap(S[lt+1], S[i]), lt←lt+1, i←i+1
5: **If** 当前 i 指向的元素>v **then** swap(S[gt-1], S[i]), gt←gt-1, i←i+1
6: If gt==i then swap(S[lt], S[p]), lt←lt-1
7: 不断递归，重复第 1-6 步

该部分核心代码如清单 2 所示。

清单 2 优化后的三路快排核心代码

```
void rand_partition_improve(int *A, int p, int r, int *m, int *n) {
	srand((unsigned)time(0));
	int v = my_generate_node(p, r + 1);
	int less = p - 1;
	int more = r;
	int idx = p;
	for (; idx<more;) {
		if (A[idx]<A[r]) {
			swap_int(&A[++less], &A[idx++]);
		}
		else  if (A[idx]>A[r]) {
			swap_int(&A[--more], &A[idx]);
		}
		else {
			idx++;
		}
	}
	swap_int(&A[idx], &A[r]);
	*m = less;
	*n = more;
}
```

(3) 数据生成

生成数据时，需要注意对生成元素进行查重，将数据重复率限定在要求范围内，该部分核心代码如清单 3 所示。

清单 3 数据生成代码

```
long my_generate_node(long p, long r) {
	return myrandom(r - p) + p;
}

bool check_repeate(vector<int> &data, int temp) {
	for (int i = 0; i < data.size(); i++) {
		if (data[i] == temp) return false;
	}
	return true;
}

void generate_node(char* init_path) {
	const char *data_path;
	long num, range_l, range_r;
	float ratio;
	map<string, string>fname;
	CParseIniFile config;
	bool flag = config.ReadConfig(init_path, fname, "0");
	if (flag) {
```

```
            num = stoi(fname["num"]);
            ratio = stof(fname["ratio"]);
            range_l = stoi(fname["range_l"]);
            range_r = stoi(fname["range_r"]);
            data_path = fname["data_path"].c_str();
      }
      else {
            cout << "Loading ini 0 error!" << endl;
            return;
      }
      srand((unsigned)time(0));
      vector <int> data;
      //generate data repeately.
      flag = false;
      int gnt_data = my_generate_node(range_l, range_r+1);
      cout << "Generating data..." << endl;
      for (int i = 0; i < num; i++) {
            Print_process(num, i, 100);
            if (i < num*ratio) {
                  data.push_back(gnt_data);
            }
            else{
                  flag = false;
                  do {
                        gnt_data = my_generate_node(range_l, range_r+1);
                         if (check_repeate(data, gnt_data)) flag = true;
                  } while (!flag);
                  data.push_back(gnt_data);
            }
      }
      cout << endl;
      unsigned seed =
chrono::system_clock::now().time_since_epoch().count();
      default_random_engine rng(seed);
      srand((unsigned)time(0));
      shuffle(data.begin(), data.end(), rng);

      FILE *fp;
      errno_t err;
      if ((err = fopen_s(&fp, data_path, "ab+")) != 0)
            printf("Open dst file failed.\n");

      cout << "Writing tf..." << endl;
      fwrite(&num, sizeof(int), 1, fp);
      for (int i = 0; i < num; ++i)
      {
            fwrite(&data[i], sizeof(int), 1, fp);
            Print_process(num, i, 100);
      }
```

```
	cout << endl;
	fclose(fp);
	cout << "Finished." << endl;
}
```

(4) 结果正确性检验

对于排序后的数组，本案例写了清单 4 所示代码检验其正确性。

清单 4 结果正确性检验代码

```
bool isSorted(int *arr, int num){
	for (int i = 0; i + 2 < num - 1; i++) {
		if (
			(arr[i] > arr[i + 1] && arr[i + 1] < arr[i + 2]) ||
			(arr[i] < arr[i + 1] && arr[i + 1] > arr[i + 2])
			) {
			return false;
		}
	}
	return true;
}
```

4. 实验结果与分析

(1) 实验参数配置

本实验将 exe 程序和 config 配置文件抽离开来，能够实现较好的交互，对配置文件的说明如清单 5 所示。用户可以在 method 处选择功能，0 是生成数据集，1 是系统的 qsort，2 是随机快排和改进后的三路快排，可以在对应的功能模块下根据提示选择。

清单 5 config.ini

```
[method]
; 0: generate node
; 1: quicksort
; 2: sort
method = 1

[0]
;生成随机点
num = 10
; 随机数的上下界
range_l = 0
range_r = 10
;重复率
ratio = 0.3
data_path = L:/why_workspace/cpl/txt/quickdata_10.tf

[1]
;总随机点，用于数据集核对
```

```
num = 1000000
;读入随机点，需要小于 num
num_chose = 1000000
data_path = L:/why_workspace/lab4/0.9/quickdata_0.9.tf
p = 0
; r 需要小于 num_chose
r = 1000000
; 是否采用三路快排
whether_improve = 0

[2]
num = 1000000
num_chose = 1000000
data_path = L:/why_workspace/lab4/0/quickdata_0.tf
```

(2)随机快排结果

通过清单 3 的代码生成 11 个数据集后，运行清单 1 的代码，得到了图 2 所示结果。

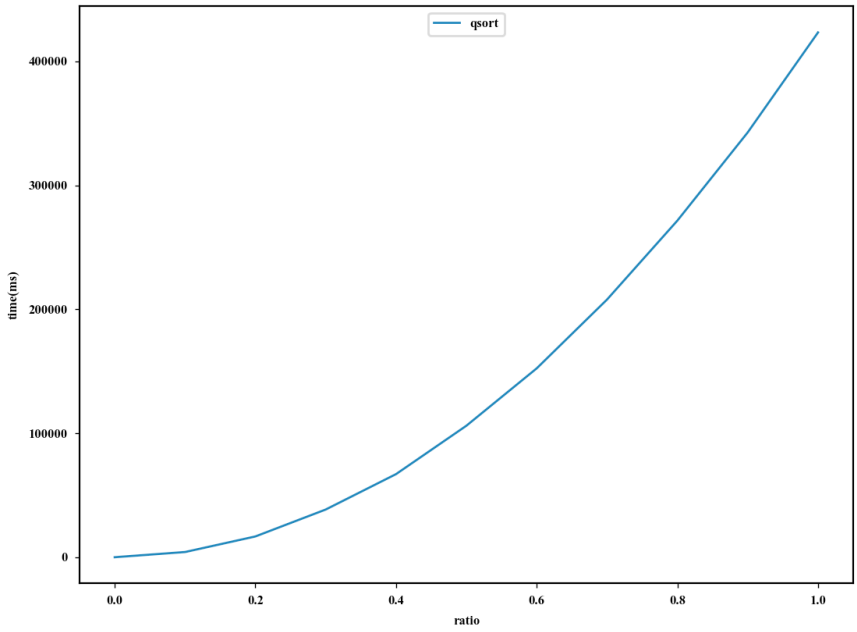

图 2 随机快排运行结果(横轴为数据集的元素重复率，纵轴为排序时间)

(3) 三路快排性能对比

运行清单 3 的代码可得到改进后的三路快排的结果，与 C++中的 qsort 快排函数进行对比，得到了图 3 所示结果。

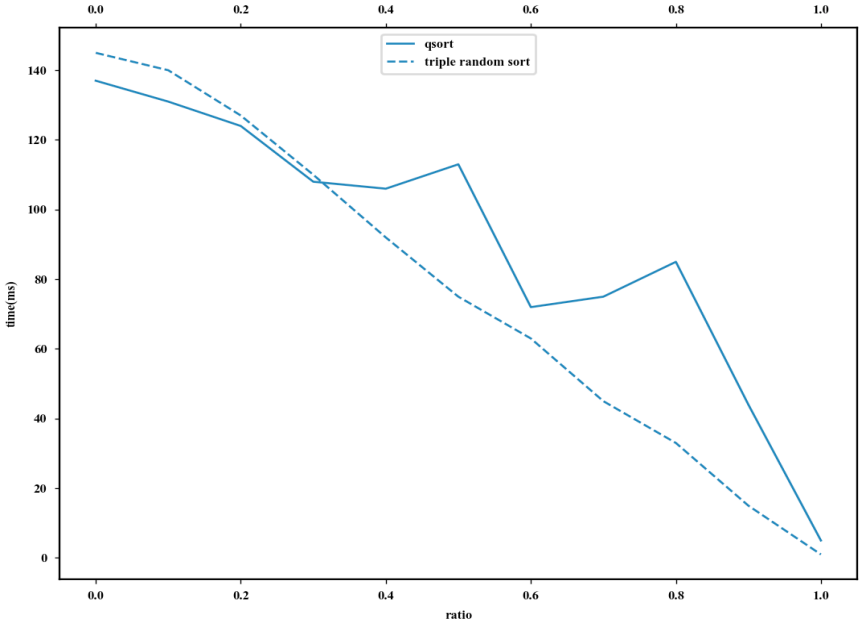

图 3 三路快排和系统库函数运行结果对比(对比的库函数为 qsort)

(4) 总结

在数据集中元素重复较少时，三路快排和随机快排的区别不明显，当重复元素较多时，运行时间和空间有显著差异。造成这种现象的原因主要是随机快排中无法处理大量的重复元素，会造成递归树不平衡，导致递归层次多，速度慢，占据空间大。再加入了=v 后，进化成三路快排，可有效解决此问题。

对比 qsort 函数，本实验改进的三路快排具有与之几乎一样的运行时间，优化结果理想，同时可以猜测该函数可能采用了类似的处理方式。

5. 实验心得

这次实验是四个实验中对我个人而言最简单的一个，主要是大名鼎鼎的快排算法写过了很多遍，因此复现和改进起来也比较容易。最后对比 C++的快排函数 qsort，发现自己写的三路快排和库函数速度几乎一样，甚至稍快，成就感满满的~