Lab 1: 快速排序算法及其优化

PB21020718 曾健斌

一、实验内容

本实验要求使用**快速排序及其优化算法**对于文件`data.txt`中的数组进行排序并输出在文件`sorted.txt`中

- 编程实现快速排序算法
- 使用**枢轴选择、混入插入排序、聚集**等手段实现对于快速排序的优化

二、快速排序的实现

本实验使用c++实现

```
QUICKSORT(A,p,r)
{
    if p < r then{
        q <- PARTITION(A,p,r);
        QUICKSORT(A,p,q-1);
        QUICKSORT(A,q+1,r);
    }
}
```

具体实现如下:

```
void
Quicksort::SimpleQuickSort_Sub(int p, int r)
{
    if(p < r){
        int q = Partition(p, r, r);
        SimpleQuickSort_Sub(p, q-1);
        SimpleQuickSort_Sub(q+1, r);
    }
}

void
Quicksort::SimpleQuickSort()
{
    SimpleQuickSort_Sub(0, length-1);
}
```

下面是PARTITION的实现

```
PARTITION(A,p,r)
{
    x <- A[r];
    i <- p - 1;
    for j = p to r - 1 do{
        if A[j] < x then{
         i <- i + 1;
         exchange A[i] with A[j];
        }
    }
    exchange A[i+1] with A[r];
    return i + 1;
}
```

转化为具体实现：

```
int
Quicksort::Partition(int p, int r, int pivotindex)
{
    int pivot = Arr[pivotindex];
    swap(Arr[r], Arr[pivotindex]);
    int i = p-1;
    for (int j=p; j<r; ++j){
        if(Arr[j] <= pivot){
            ++i;
            swap(Arr[i], Arr[j]);
        }
    }
    swap(Arr[i+1], Arr[r]);
    return i+1;
}
```

三、快速排序的优化

优化算法主要有以下几种方式：

- 枢轴选择：固定枢轴，随机枢轴，三值取中
- 在几乎有序的时候使用插入排序
- 聚集元素

1、枢轴选择

固定枢轴

在固定枢轴中我们恒定选择划分的最后一个元素为枢轴，但是这对于有序度较高的数组会造成分割极度不均衡的情况，从而导致算法的时间复杂度逼近其最坏时间复杂度$O(n^2)$

二中的实现即是使用了固定枢轴

随机枢轴

为了解决上述问题，可以使用随机选择枢轴的方法，虽然这可能会导致某些情况算法变慢，但从总体的期望的角度来看，随机枢轴优化了算法

具体实现如下：

```
int
Quicksort::RandomPartition(int p, int r)
{
    int randompivot = p + rand() % (r - p + 1);
    return Partition(p, r, randompivot);
}

void
Quicksort::RandomPivot_Sub(int p, int r)
{
    if(p < r){
        int q = RandomPartition(p, r);
        RandomPivot_Sub(p, q-1);
        RandomPivot_Sub(q+1, r);
    }
}

void
Quicksort::RandomPivot()
{
    srand(seed);                                        //1
    RandomPivot_Sub(0, length-1);
}
```

`//1`处`seed`为类中定义的字段，可通过类方法`SetSeed(int)`调节

三值取中

对于快速排序而言，每次划分得越均匀效果越好，因此中位数作为枢轴是最好的选择，但是得到中位数时间复杂度过高，故使用三值取中的方式来贴近这种效果

具体实现方式是以`A[p]`，`A[r]`，`A[(p+r)/]`中的中值为枢轴，代码如下：

```
int
Quicksort::MedianofThree(int left, int right)
{
    if(left == right) return left;
    int mid = left + (right - left)/2;
    if(Arr[left] < Arr[mid]){
        if(Arr[mid] < Arr[right]){
            return mid;
        }
        else{
            if(Arr[left] < Arr[right]){
                return right;
            }
            else return left;
        }
```

```
    }
    else{
        if(Arr[mid] > Arr[right]){
            return mid;
        }
        else{
            if(Arr[left] < Arr[right]){
                return left;
            }
            else return right;
        }
    }
}

int
Quicksort::PartitionMOT(int p, int r)
{
    int pivot = MedianofThree(p, r);
    return Partition(p, r, pivot);
}

void
Quicksort::MOTSort_Sub(int p, int r)
{
    if(p < r){
        int q = PartitionMOT(p, r);
        MOTSort_Sub(p, q-1);
        MOTSort_Sub(q+1, r);
    }
}

void
Quicksort::MOTSort()
{
    MOTSort_Sub(0, length-1);
}
```

2、在几乎有序的时候使用插入排序

在输入数组已经“几乎有序”时，使用插入排序速度更快。因此，修改`PARTITION`使得在子数组长度小于 k 时调用插入排序：

```
void
Quicksort::QuickInsertSort_Sub(int p, int r, int k)
{
    if(k < r - p){
        int q = Partition(p, r, MedianofThree(p, r));
        QuickInsertSort_Sub(p, q-1, k);
        QuickInsertSort_Sub(q+1, r, k);
    }
    else{
        InsertSort(p, r);
    }
}

```

```
void
Quicksort::QuickInsertSort(int k)
{
    QuickInsertSort_Sub(0, length - 1, k);
}
```

3、聚集元素

在一次分割之后，将与本次枢轴相等的元素聚集在一起，再次进行分割时，不再对聚集过的元素进行分割：

```
void
Quicksort::GatherPartition(int p, int r, int pivotindex)
{
    int pivot = Arr[pivotindex];
    swap(Arr[r], Arr[pivotindex]);
    int ecount = 0;
    int i = p-1;
    int j = p;
    while(j < r - ecount){
        if(Arr[j] == pivot){
            ++ecount;
            swap(Arr[j], Arr[r-ecount]);
        }
        if(Arr[j] < pivot){
            ++i;
            swap(Arr[j], Arr[i]);
        }
        ++j;
    }

    int k = i+1;
    for(j=0; j <= ecount; ++k, ++j){
        Arr[r-j] = Arr[k];
        Arr[k] = pivot;
    }
    Pgather.push(i+1);                                  //1
    Pgather.push(i+1+ecount);
}

void
Quicksort::GatherSort_Sub(int p, int r)
{
    if(p < r){
        GatherPartition(p, r, MedianofThree(p, r));
        GatherSort_Sub(Pgather.top()+1, r);
        Pgather.pop();
        GatherSort_Sub(p, Pgather.top()-1);
        Pgather.pop();
    }
}

void
Quicksort::GatherSort()
{
```

```
    GatherSort_Sub(0, length-1);
}
```

`//1`为类中的一个`stack<int>`字段

四、实验结果分析

本实验定义了一个用于计时的`Timer`类进行时间分析，以下是其类方法

```
void
Timer::start()
{
    start_time = chrono::high_resolution_clock::now();
}

void
Timer::stop()
{
    end_time = chrono::high_resolution_clock::now();
}

double
Timer::elapsedMilliseconds()
{
    chrono::duration<double, milli> elapsed = end_time - start_time;
    return elapsed.count();
}
```

对每一种方法作50次排序得到平均结果，下例：

```
for(int i = 1; i < 50; ++i){
    qs = Quicksort(test2.size(), test2);
    qs.SetSeed(static_cast<unsigned>(time(nullptr)));
    tm.start();
    qs.Sort(20);
    tm.stop();
    Time += tm.elapsedMilliseconds();
}
cout << "Sort: " << Time/50.0 << endl;
```

得到以下结果：

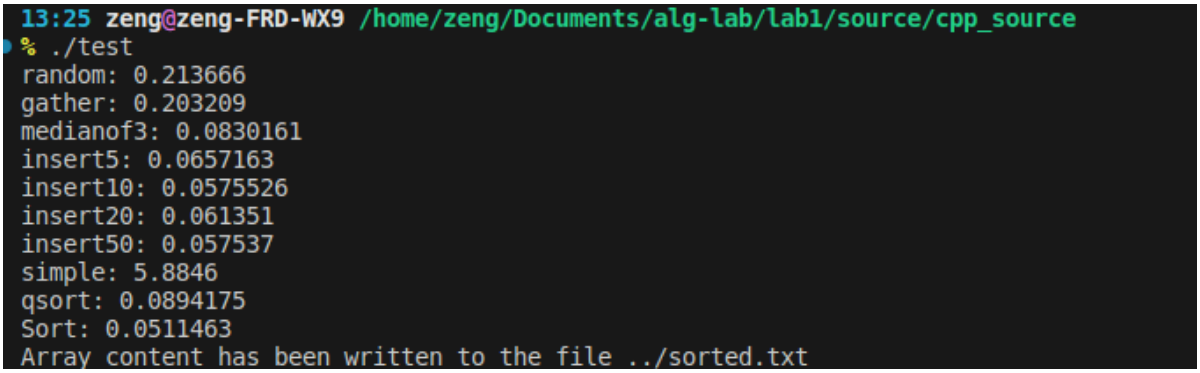
```
12:24 zeng@zeng-FRD-WX9 /home/zeng/Documents/alg-lab/lab1/source/cpp_source
% ./test
random: 23.6199
gather: 29.2473
medianof3: 22.6446
insert5: 19.9618
insert10: 19.6342
insert20: 19.2363
insert50: 21.119
simple: 21.9962
qsort: 16.05
Sort: 10.0778
Array content has been written to the file ../sorted.txt
```

分别为随机枢轴、聚集、三值取中、插入排序(k=5)、插入排序(k=10)、插入排序(k=20)、插入排序(k=50)、固定枢轴、C++自带的`sort()`、三值取中 + 插入排序 + 聚集

似乎三值取中并没有带来很明显的提高，而随机枢轴甚至使排序速率降低了，实际上应该是与给定的排序数据顺序较乱导致固定排序的缺点未被体现出来

可以看到三**值取中 + 插入排序 + 聚集**对于排序速度有很大的提升

此处再使用一个已经排好序的数据集再次进行测试：



注：本实验中除了固定枢轴的原始算法，其他实现都使用了三值取中或随机枢轴

可以看到对于一个有序的数据集，固定枢轴的复杂度达到了恐怖的$O(n^2)$，而插入排序的 k 值越大，算法则越快。可以看到三**值取中 + 插入排序 + 聚集**还是取得了相当好的效果，并且较于未优化的快排序算法，该算法能应对更多的情况。

五、实验总结

本实验完成了快速排序算法的实现与优化，在实验过程中有以下收获：

- 通过对不同数据集的测试，深刻体会到固定枢轴的劣势
- 对于聚集以及与插入排序的混用给予了我很多使用的排序算法优化思路