Experiment-Quick Sort

PB18010496 杨乐园

Introduction

编程实现基本快速排序（默认基准选定最后一个元素$A[r]$），并实现快速排序的优化，并与其他排序方法作对比分析。

Algorithm

1. 基本快排

基本的快速排序，通过采用分治思想，调用$partition_last$函数对输入的数组$A[p,..,r]$划分为两个子数组$A[p,...,q-1]$和$A[q+1,...,r]$，使得$A[p,...,q-1]$中的元素均小于等于$A[q]$，而$A[q+1,...,r]$中的元素均大于等于$A[q]$，从而对子数组递归调用$quicksort$函数即可。如下为关键代码：

```
int partition_last(vector<int>& A, int p, int r)
{
    int x = A[r];
    int i = p - 1;
    int temp = 0;
    for (int j = p; j <= r - 1; j++)
    {
        if (A[j] <= x)
        {
            i = i + 1;
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }
    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;
    return i + 1;
}

void quicksort(vector<int>& A, int p, int r)
{
    if (p < r)
    {
        int q = partition_last(A, p, r);
        quicksort(A, p, q - 1);
        quicksort(A, q + 1, r);
    }
}
```

2. 快排的优化之基准的选择

对于上述最基本的快速排序代码我们可以看到，快速排序运行时间与划分是否对称有直接关系。最坏情况下，每次划分过程中产生的两个子区域分别包含$n-1$个元素和1个元素，这样其时间复杂度即会达到$O(n^2)$。在最好情况下，每次划分所取的基准都恰好是区间的中指，也即每次划分都产生两个大小为$\frac{n}{2}$的区域，此时，快速排序便达到了时间复杂度为$O(nlogn)$。所以基准的选择对于快排而言，也就变得十分重要，对快排的优化，某种程度上也可以看成是选择足够好的基准。

对于上面最基准的快排，采用的是固定基准，即$A[r]$，我们也可以采取随机基准。对于传入的数组$A[p,..,r]$，通过生成$[p,r]$区间内随机整数q，从而以$A[q]$作为基准进行区域划分，由于$C++$没有生成具体区间内随机整数的函数，我们通过数学计算给出，即$(rand()\%(r-p+1))+p$，如下为划分的代码：

```
int randomized_partition(vector<int>& A, int p, int r)
{
    int i = (rand() % (r - p + 1)) + p; //生成[p,r]间的随机数;
    int temp = A[r];
    A[r] = A[i];
    A[i] = temp;
    return partition_last(A,p,r);
}
```

除了固定基准与随机基准，还有一种"三数取中"的基准，即对传入的数组$A[p,..,r]$，取$A[p],A[\lfloor\frac{(p+r)}{2}\rfloor],A[r]$的中位数作为此次划分的基准。需要注意的是，我们需要自己写取中位数的函数，由于三个数的特殊性，我们采取三个if语句即可给出中位数的取法，具体如下所示：

```
int partition_mid(vector<int>& A, int p, int r)
{
    int a = A[p], b = A[floor((p + r) / 2)], c = A[r];
    int x = 0;

    if ((a - b) * (a - c) <= 0)
        x = a;
    if ((b - a) * (b - c) <= 0)
        x = b;
    if ((c - a) * (c - b) <= 0)
        x = c;

    int i = p - 1;
    int temp = 0;
    for (int j = p; j <= r - 1; j++)
    {
        if (A[j] <= x)
        {
            i = i + 1;
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }
    temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;
    return i + 1;
}
```

3. 快排的优化之结合插入排序

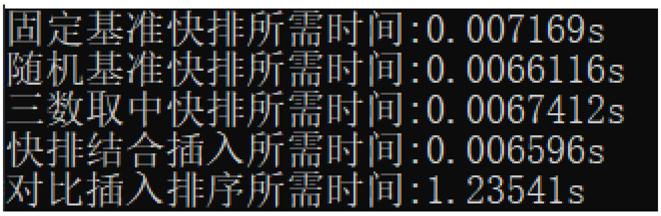
首先我们知道，当输入的一组数据几乎有序时，使用插入排序时速度将会非常的快，正是基于这一点，我们可以对快速排序进行优化提速。我们回看快速排序，由于递归的调用，快速排序会逐渐减小数组规模直至进行到长度仅为1的情况，由于上述插入排序的特点，我们选择在合适的递归层数调用插入排序，也即当对一个长度小于k的数组调用快速排序时让函数不做任何排序就返回，当上层所有的快排进行完毕后，对整个数组调用插入排序来完成最后的排序任务。注意到，由于只有最多k个范围内的无序性，所以排序不会是$O(n^2)$的时间复杂度。关键代码如下：

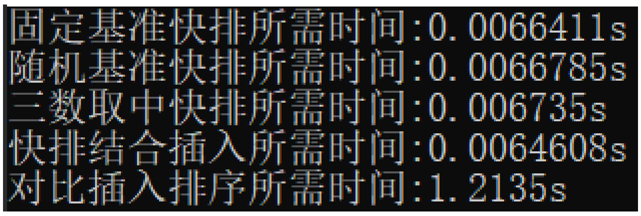
```
//插入排序
void insertsort(vector<int>& A)
{
    int key = 0, i = 0;
    for (int j = 1; j < A.size(); j++)
    {
        key = A[j];
        i = j - 1;
        while (i >= 0 && A[i] > key)
        {
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
}

//区间长度<=k时不做快排，直接返回；
void quicksort_k(vector<int>& A, int p, int r, int k)
{
    if (p<r && r - p >= k)
    {
        int q = partition_last(A, p, r);
        quicksort_k(A, p, q - 1, k);
        quicksort_k(A, q + 1, r, k);
    }
}

//几乎有序的快排；
void quicksort_insert(vector<int>& A, int p, int r, int k)
{
    quicksort_k(A, p, r, k);
    insertsort(A);
}
```

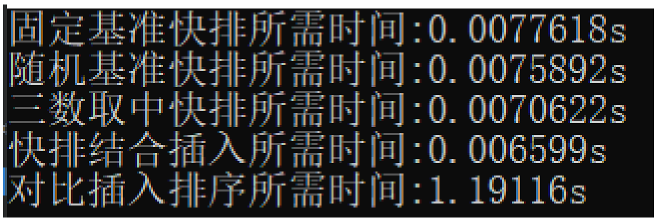
Results

我们对同一组测试数据（100000个）给出三次测试，分别记录相应的时间：

	$First$	$Second$	$Third$
固定基准$A[r]$	$0.007169s$	$0.0066411s$	$0.0077618s$
随即基准	$0.0066116s$	$0.0066785s$	$0.0075892s$
三数取中	$0.0067412s$	$0.006735s$	$0.0070622s$
结合插入排序	$0.006596s$	$0.0064608s$	$0.006599s$
插入排序	$1.23541s$	$1.2135s$	$1.19116s$







通过对上述运行结果的比较，我们可以看到：相对于插入排序$O(n^2)$，快速排序的速度明显更快，几乎是快了200倍，足见快排时间复杂度的优势；而对快排的优化方面，随机基准则并不总是达到了优化效果，如第二组测试，这也体现了随机性；三数取中的基准选择亦然，在第二组测试中仍没有达到所需要的优化效果；而结合插入排序时，我们可以看到三次都达到了优化效果，可见这个优化处理相对成功。

不过总体而言，快速排序给出了一种相对时间复杂度较低的排序方法，其代码书写简单，效率高，是一种十分优秀的排序策略。

Code

具体完整代码，参看附件文件加$QuickSort$。