作业7 中位数

> 比较 Quick Select 与 Linear Select 的性能差异以及一些参数对算法的影响

实现两种算法

- 首先实现快速选择的方法

```
// 返回数组中，从array里面第k小的元素，left和right仅仅是为了递归调用
// left和right仅仅代表搜索的范围，每次搜索的第k小的元素，是在整个数组里面的第k小的元素
// 而不是在left和right之间的第k小的元素！！！（和后面的有一些区别）
// k是从0开始的!
int QuickSelect(vector<int> &array, int left, int right, int k){
    if(left == right){
        return array[left];
    }

    int pivot = array[left];
    int i = left + 1;
    int j = right;
    while(i <= j){
        while(i <= j && array[i] <= pivot){
            i++;
        }
        while(i <= j && array[j] > pivot){
            j--;
        }
        if(i < j){
            swap(array[i], array[j]);
        }
    }
    swap(array[left], array[j]);

    if(j == k){
        return array[j];
    }else if(j < k){
        return QuickSelect(array, j + 1, right, k);
    }else{
        return QuickSelect(array, left, j - 1, k);
    }
}
```

- 然后实现LinearSelect。相比快速选择要更复杂一些

```
// 返回数组中，从left到right区间内第k小的元素
// k是从0开始的!
int LinearSelect(vector<int> &array, int left, int right, int k){
    // 如果区间长度小于等于Q，直接快速排序
    if(right - left + 1 <= Q){
        sort(array.begin() + left, array.begin() + right + 1);
```

```
            return array[left + k];
        }

        // 将A均匀地划分为n/Q个子序列，各含Q个元素；将返些中位数组成一个序列seqAll;
        vector<int> seqAll;

        for (size_t i = 0; i < array.size(); i+= Q)
        {
            size_t result = QuickSelect(array, i, min(i + Q - 1, array.size() - 1), Q / 2);
            seqAll.push_back(result);
        }

        // 递归地求出seqAll的中位数
        int pivot = LinearSelect(seqAll, 0, seqAll.size() - 1, seqAll.size() / 2);
        // 根据其相对于M的大小，将A中元素分为三个子集：L（小于）、E（相等）和G（大于）；
        vector<int> LGroup;
        vector<int> EGroup;
        vector<int> GGroup;

        for (size_t i = 0; i < array.size(); i++)
        {
            if(array[i] < pivot){
                LGroup.push_back(array[i]);
            }else if(array[i] == pivot){
                EGroup.push_back(array[i]);
            }else{
                GGroup.push_back(array[i]);
            }
        }

        // 如果k小于L，递归地在L中寻找第k小元素
        if (k < LGroup.size())
            return LinearSelect(LGroup, 0, LGroup.size() - 1, k);
        // 如果k大于L和E，返回M
        else if (k < LGroup.size() + EGroup.size())
            return pivot;
        // 如果k大于L和E，递归地在G中寻找第k - L.size() - E.size()小元素
        else
            return LinearSelect(GGroup, 0, GGroup.size() - 1, k - LGroup.size() - EGroup.size());
    }

```

- 为了简要的验证算法的设计是否正确，我定义了一个正确性测试函数
- 这个函数的逻辑是生成一些随机数，然后放入两个数组用来测试。
- 一个数组用来测试搜索，另一个数组需要排序用来检测答案是否正确。
- 然后遍历这个数组里面的第`k`（k从0-数组大小）大的元素，配合验证数组，检查发现全部通过

```
int correctTest(int length){
    vector<int> tmpData;
```

```cpp
    vector<int> dataForVerify;
    srand(time(NULL));
    for(int i = 0; i < length; i++){
        int num = rand();
        tmpData.push_back(num);
        dataForVerify.push_back(num);
    }

    sort(dataForVerify.begin(), dataForVerify.end());

    for(int i = 0; i < length; i++){
        int result = LinearSelect(tmpData, 0, tmpData.size() - 1, i);
        if(result != dataForVerify[i]){
            cout << "LinearSelect Error!" << endl;
            return 1;
        }
        else if(i % 100 == 0)
            cout << "LinearSelect OK!" << endl;
    }

    for(int i = 0; i < length; i++){
        int result = QuickSelect(tmpData, 0, tmpData.size() - 1, i);
        if(result != dataForVerify[i]){
            cout << "QuickSelect Error!" << endl;
            return 1;
        }
        else if(i % 100 == 0)
            cout << "QuickSelect OK!" << endl;
    }
    return 0;
}
```

- 测试发现全部通过

数据规模和有序性的影响

- 为了测试数据规模和有序性的影响设计了实验
- 随机数生成的函数如下，为了便于测试，增加了policy用来确定是递增的数据还是递减数据

```cpp
// 随机产生数据 policy 为 0 时，数据为随机数；为 1 时，数据需要升序排序
// 为2时，数据需要降序排序
void generateData(int policy, int length){
    srand(time(NULL));
    dataAll.clear();
    for(int i = 0; i < length; i++){
        dataAll.push_back(rand());
    }

    if(policy == 1){
        sort(dataAll.begin(), dataAll.end());
    }
    else if(policy == 2){
```

```
            sort(dataAll.begin(), dataAll.end());
            reverse(dataAll.begin(), dataAll.end());
        }
    }
```

- 为了保证不同测试之间的控制变量，我选择对于所有的测试都是测试64次（因为最小的数据集合就是64长度）
- 此外，测试选择的k是依次是$\frac{i \times Length}{64}$
- 所有的测试的数据长度都是64的倍数
- 如下所示是统计线性选择的函数的时间。

```
/**
 * @param length: 数据长度
 * @param ifOrder :0: 无序, 1: 升序, 2: 降序
*/
double performanceTest_LinearSelect(int length, int ifOrder){
    generateData(ifOrder, length);
    clock_t start, end;
    start = clock();
    for(int i = 0; i < length; i+= length / 64){
        int result = LinearSelect(dataAll, 0, dataAll.size() - 1, i);
    }
    end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC;
}
```

- 下面的结果是linearSelect的情况

数据量	乱序	递增	递减
64	3e-05	1.9e-05	2.1e-05
1024	0.027812	0.028811	0.020593
4096	0.103519	0.342241	0.127836
8096	0.320496	2.14541	0.69423

- 下面的排序结果是quickSelect的情况

数据量	乱序	递增	递减
64	0.000255	0.000244	0.000247
1024	0.052197	0.052619	0.05331
4096	0.796694	0.84217	0.852486
8096	3.19663	3.39107	3.40953

> 首先我们来对比快速选择和线性选择：

- 首先查阅教材，我发现对于快速选择排序的性能解释如下：尽管内循环仅需O(hi - lo + 1)时间，但很遗憾，外循环的次数却无法有效控制。与快速排序算法一样，最坏情况下外循环需执行O(n)次（习题[12-11]），总体运行时间为O(n2)。
- 在最坏情况下，每一次随机选取的候选轴点pivot = A[lo]都不是查找的目标，而且偏巧 就是当前的最小者或最大者。于是，对向量的每一次划分都将极不均匀，其中的左侧或右侧子向 量长度为0。如此，每个元素都会被当做轴点的候选，并执行一趟划分，累计O(n)次。
- 所以可以看到，因为我测试的时候选取的k分布比较广，而不是集中的选择了某几个区域。所以可导致表格里面中快速选择的时间要普遍高于linearSelect的情况
- 此外，如果纵向对比时间消耗，可以发现linearSelect比较**稳定**，什么叫稳定呢，就是增加的很有规律。数据量增加大约4倍的时候，时间同样增加大约四倍，但是quickSelect就不一样了，比较第二行、第三行数据数据量增加4倍，时间增加了20倍

> 然后我们分析一下增序、降序、乱序的影响

- 先说QuickSort，其实从大局来看顺序、乱序对于QuickSort影响尽管有，但是没有那么明显
- 显然如果是顺序递增的时候，下面的循环将会是一路畅通（外循环只用跑一次）。反之如果是顺序递减的时候，外循环要跑大约n/2次。这也造成时间上面递减的比递增的要稍微慢一些
- 然后，我认为因为有序的时候，每次取到的pivot都很极端（不管是最小值还是最大值，都是个极端值，但是无序的时候取到的就可能是正中间的啊某些值，这对于搜索的第k大更加友好）

```
      while(i <= j){
          while(i <= j && array[i] <= pivot){
              i++;
          }
          while(i <= j && array[j] > pivot){
              j--;
          }
          if(i < j){
              swap(array[i], array[j]);
          }
      }
```

- 再说LinearSelect，分析一遍代码，并没有看到对于因为数据大小不同产生明显影响差异的代码段。维度可能产生差异的就是递归阶段

```
        // 如果k小于L，递归地在L中寻找第k小元素
        if (k < LGroup.size())
            return LinearSelect(LGroup, 0, LGroup.size() - 1, k);
        // 如果k大于L和E，返回M
        else if (k < LGroup.size() + EGroup.size())
            return pivot;
        // 如果k大于L和E，递归地在G中寻找第k - L.size() - E.size()小元素
        else
            return LinearSelect(GGroup, 0, GGroup.size() - 1, k - LGroup.size() - EGroup.size());
```

- 仔细斟酌可以发现，LinearSelect每次选择的参照对象是中位数！而不是QuickSelect里面随便捞了个第0个元素就是的。
- 我尝试查看每个函数调用的次数，然后发现递增的时候调用的次数4411（8096数据那一组），而递减的时候调用的次数1642，整整差了4倍。我试图分析了一下调用的规律，但是并没有找到什么头绪，如果硬要比较递增和递减的差异。
- 如果对比顺序和乱序，规律还是比较明显的，乱序的情况下，更容易遇到下面递归终止的情况，而顺序执行的时候除非要找的就是中位数，否则基本上都要往下一步递归。

```
    else if (k < LGroup.size() + EGroup.size())
        return pivot;
```

> 最后分析一下数据量的情况

- 数据量的话基本上都是越多消耗的时间越大
- 随着数据量增加，linearSelect增加的相对更加稳定（数据量增加多少，时间同步增加）但是quickselect的增长就没有那么稳定

Q的选择

- 关于Q的选择，我测试了`int QTest[] = {64, 128, 256, 512, 1024};`的情况
- 数据量保持为4096不变化。得到的结果如下

```
64,0.153615
128,0.104005
256,0.067815
512,0.048381
1024,0.065871
```

- 可以观察到随着数据量的增加，消耗的时间先减小、然后再增加，分析原因如下：
- Q太小的时候，分组太多了，要把这么多组里面的中位数全部找出来，本身就需要消耗时间，主要因素都是分组过多带来的中位数查找开销，所以要的时间增加
- Q太大的时候，相当于没有分组，比如Q如果大到了超过数组长度，相当于之间找了中位数然后判断，没有体现线性选择的意义了
- 所以Q过大或者过小都不好，为了让性能最优，可能需要一些数学推导和证明，才能找到最优的Q大小。