数据结构与算法 II 上机实验 (10.23)

中国人民大学　信息学院　崔冠宇　2018202147

上机题 1　实现最坏情况为线性时间的选择第 k 大数算法。

一、问题描述

1. 输入：序列 $< a_1, a_2, \cdots, a_n >$，正整数 $k(1 \leq k \leq n)$。
2. 输出：序列中第 k 大数 a'_k，即它满足 $a'_1 \geq a'_2 \geq \cdots \geq a'_k \geq \cdots a'_n$，其中各 a'_i 是各 a_j 的重排。

二、算法基本思路

首先，题目要求的是实现第 k 大数算法，我们可以将其转化为求解第 $n-k+1$ 顺序统计量，从而利用线性时间算法 (BFPRT) 解决。

其中 BFPRT 算法的基本思想是：

1. 将数组按五个一组划分；
2. 求出每组中位数，将其放到数组前面；
3. 再对数组前边中位数部分调用 BFPRT，求出各中位数的中位数，选定其作为主元放在数组第一个位置；
4. 运行划分算法，得到主元的相对位次；
5. 根据其与所需位次的相对大小，对左右两部分递归调用 BFPRT。

下面给出 BFPRT 算法的伪代码:

```
BFPRT(A, l, r, i):
// 仅一个元素
if l == r
    return l
// 5个一组
groups = ceil((r - l + 1) / 5)
// 每组找到中位数，放到前面 (n/5)O(1)=O(n)
for i = 1 to groups
    mid = findMid5(A, l + 5 * i, min(l + 5 * i + 4, r))
    swap(A[l + i, A[mid]])
// 中位数的中位数 T(n/5)
midmid = BFPRT(A, l, l + groups - 1, (groups + 1) / 2)
// 放到最前，便于作为主元划分
```

```
swap(A[l], A[midmid])
// 划分所得下标 O(n)
partitionIndex = Partition(A, l, r)
// 主元相对排名
relativeRank = partitionIndex - l + 1
// 恰好找到
if relativeRank == i
    return partitionIndex
// T(7n/10 + 6) ——— 《算法导论》P123
// 主元排名偏大，在左边找
if relativeRank > i
    return BFPRT(A, l, partitionIndex, i)
// 右边找
else return BFPRT(A, partitionIndex + 1, r, i - relativeRank)
```

程序设计思路：

1. 首先应该实现 `findMid5`，以查找每组中位数，我决定将 5 个数排序，直接找到中位数；
2. 其次应该实现 `Partition`，以数组第一个数为主元，划分数组，使得小于主元的数放在左边，大于主元的数放在右边，并返回主元下标；
3. 根据伪代码，用 BFPRT 算法实现 `Select` 函数，找出数组中第 i 顺序统计量；
4. 读入老师给出的数据，调用 `Select`，找出第 $i = n - k + 1$ 顺序统计量即可；
5. 在老师给的数据上，对比线性时间算法和排序实现的 $\Theta(n \log n)$ 平凡算法的运行时间；
6. 生成数千万个数的大数据集，再次对比两种算法运行时间。

三、算法复杂性分析

为了分析算法的时间复杂度，设输入规模为 n 时，算法的运行时间为 $T(n)$。

1. 算法中的 `findMid5` 是找出 5 个数的中位数，故为 $\Theta(1)$ 的；因为有 $\lceil n/5 \rceil$ 个组，所以是 $\Theta(n)$ 的；
2. 递归调用求中位数的中位数时，显然所用时间是 $T(n/5)$；
3. `Partition` 是以第一个数为主元，进行快排划分，故为 $\Theta(n)$ 的；
4. 根据《算法导论》P123 的讨论，最坏情况下，剩余的规模为 $T(7n/10 + 6)$。

所以根据分析可以写出 $T(n) \leq T(n/5) + T(7n/10 + 6) + \Theta(n)$。利用代入法，可以解出 $T(n) = O(n)$。

四、程序源代码

线性时间第 k 大数: **select.cpp**:

```cpp
#include <iostream>
#include <fstream>
#include <utility>
#include <ctime>
#include <cmath>
#include <algorithm>
#include <string>
#include <random>
#include <vector>

// 功能: 对于每组数 arr[l...r] (小于5个), 返回中位数坐标
template <typename T>
size_t findMid5(std::vector<T> & arr, size_t l, size_t r)
{
    std::sort(arr.begin() + l, arr.begin() + r + 1);
    return l + (r - l) / 2;
}

// 功能: 按第一个元素为主元划分 arr[l...r], 小的在左边, 返回主元划分后坐标
template <typename T>
size_t Partition(std::vector<T> & arr, size_t l, size_t r)
{
    size_t low = l, high = r;
    // 第一个元素为主元
    T pivot = arr[low];
    while(low < high)
    {
        // 从后面开始找到第一个小于pivot的元素, 放到low位置
        while(low < high && arr[high] >= pivot)
        {
            high--;
        }
        arr[low] = arr[high];
        // 从前面开始找到第一个大于pivot的元素, 放到high位置
        while(low < high && arr[low] <= pivot)
```

```
36          {
37              low++;
38          }
39          arr[high] = arr[low];
40      }
41      // 最后枢纽元放到low的位置
42      arr[low] = pivot;
43      return low;
44  }
45
46  // 功能：从 arr 数组 [l, r] 下标范围内选出第 i 顺序统计量所在的下标
47  // 时间复杂度：T(n)=O(n)，但常数可能较大
48  template <typename T>
49  size_t SelectIndex_By_BFPRT(std::vector<T> & arr, size_t l, size_t r, size_t k)
50  {
51      assert(l <= r && k >= 1 && k <= r - l + 1);
52      // 仅一个元素
53      if(l == r)
54      {
55          return l;
56      }
57      // 比较少，直接排序
58      if(r - l + 1 <= 140)
59      {
60          std::sort(arr.begin() + l, arr.begin() + r + 1);
61          return l + k - 1;
62      }
63      // 5个一组，共多少组
64      size_t groupCount = std::ceil(double(r - l + 1) / 5);
65      // 每组内……
66      for(size_t i = 0; i < groupCount; i++)
67      {
68          // 找到中位数下标
69          size_t midIndex = findMid5(arr, l + 5 * i, std::min(l + 5 * i + 4, r));
70          // 交换每组中位数到前边
71          std::swap(arr[l + i], arr[midIndex]);
72      }
```

```
73      // 找到中位数的中位数
74      size_t midmidIndex = SelectIndex_By_BFPRT(arr, l, l + groupCount - 1, (
        groupCount + 1) / 2);
75      // 把中位数的中位数放到前边，方便调用 Partition
76      std::swap(arr[l], arr[midmidIndex]);
77      // 按主元划分后主元的下标
78      size_t partitionIndex = Partition(arr, l, r);
79      // 主元的相对排名=按主元划分主元的下标-l+1
80      size_t relativeRank = partitionIndex - l + 1;
81      if(relativeRank == k)
82      {
83          return partitionIndex;
84      }
85      // 主元的相对排名大于要找的，在左边找
86      if(relativeRank > k)
87      {
88          return SelectIndex_By_BFPRT(arr, l, partitionIndex, k);
89      }
90      // 主元的相对排名小于要找的，在主元右边（不含）找
91      else return SelectIndex_By_BFPRT(arr, partitionIndex + 1, r, k - relativeRank);
92  }
93
94  // 功能：从 arr 数组 [l, r] 下标范围内选出第 i 顺序统计量
95  // 时间复杂度：T(n)=O(n)，但常数可能较大
96  template <typename T>
97  T Select_By_BFPRT(std::vector<T> & arr, size_t l, size_t r, size_t k)
98  {
99      return arr[SelectIndex_By_BFPRT<T>(arr, l, r, k)];
100 }
101
102 // 功能：从 arr 数组 [l, r] 下标范围内选出第 i 顺序统计量
103 // 时间复杂度：T(n)=O(n log n)
104 template <typename T>
105 T Select_By_Sort(std::vector<T> & arr, size_t l, size_t r, size_t k)
106 {
107     assert(k >= 1);
108     std::sort(arr.begin() + l, arr.begin() + r + 1);
```

```
109     return arr[l + k - 1];
110 }
111
112 int main(int argc, char * argv[])
113 {
114     // 准备老师给的文件
115     std::cout << "准备测试数据——老师给的数据集" << std::endl;
116     std::cout << "请输入文件路径:";
117     std::string path;
118     std::cin >> path;
119     // 打开文件
120     std::fstream file(path);
121     // vector 存所需数据
122     std::vector<int> vec;
123     int readbuf = 0;
124     // 直到文件末尾
125     while(!file.eof())
126     {
127         // 读一个插入一个
128         file >> readbuf;
129         vec.push_back(readbuf);
130     }
131     // 最后一行
132     vec.pop_back();
133     // 为了测试排序方法
134     std::vector<int> vecForSort = vec;
135
136     // 输入i
137     std::cout << "请输入 k : ";
138     size_t k = 0;
139     std::cin >> k;
140
141     // 线性时间方法
142     std::cout << "选择算法(线性时间实现):" << std::endl;
143     std::clock_t begin = std::clock();
144     std::cout << "第 " << k << " 大元素是:" <<
145         Select_By_BFPRT<int>(vec, 0, vec.size() - 1, vec.size() - k + 1) << std::
```

```
    endl;
146     std::clock_t end = std::clock();
147     std::cout << "用时:" << double(end - begin) / CLOCKS_PER_SEC
148         << "秒." << std::endl;
149 
150     // 对比排序方法
151     std::cout << "选择算法(排序实现):" << std::endl;
152     begin = std::clock();
153     std::cout << "第 " << k << " 大元素是:" <<
154         Select_By_Sort<int>(vecForSort, 0, vecForSort.size() - 1, vecForSort.size()
    - k + 1) << std::endl;
155     end = std::clock();
156     std::cout << "用时:" << double(end - begin) / CLOCKS_PER_SEC
157         << "秒." << std::endl;
158 
159     std::cout << std::endl;
160 
161     // 清空, 准备增加规模
162     vec.clear();
163     vecForSort.clear();
164 
165     // 超大规模数据
166     const unsigned long long N = 50000000;
167     std::cout << "准备测试数据——超大规模(" << N << "个数据)数据集" << std::endl;
168     std::cout << "(可能较慢, 请耐心等待)" << std::endl;
169     for(unsigned long long i = 0; i < N; i++)
170     {
171         vec.push_back(i);
172     }
173     std::random_device rd;
174     std::mt19937 g(rd());
175     // 随机打乱
176     std::shuffle(vec.begin(), vec.end(), g);
177     vecForSort = vec;
178 
179     // 输入i
180     std::cout << "请输入 k : ";
```

```
181     std::cin >> k;
182 
183     // 线性时间方法
184     std::cout << "选择算法(线性时间实现):" << std::endl;
185     begin = std::clock();
186     std::cout << "第 " << k << " 大元素是:" <<
187         Select_By_BFPRT<int>(vec, 0, vec.size() - 1, vec.size() - k + 1) << std::
    endl;
188     end = std::clock();
189     std::cout << "用时:" << double(end - begin) / CLOCKS_PER_SEC
190         << "秒." << std::endl;
191 
192     // 对比排序方法
193     std::cout << "选择算法(排序实现):" << std::endl;
194     begin = std::clock();
195     std::cout << "第 " << k << " 大元素是:" <<
196         Select_By_Sort<int>(vecForSort, 0, vecForSort.size() - 1, vecForSort.size()
    - k + 1) << std::endl;
197     end = std::clock();
198     std::cout << "用时:" << double(end - begin) / CLOCKS_PER_SEC
199         << "秒." << std::endl;
200     return 0;
201 }
```

五、运行结果截图

运行 select.cpp，可以发现

1. 在老师给的 5000 个数的小数据集上，线性时间的算法竟然被排序的算法打败了，我猜测是因为线性时间算法的常数系数较大，导致在小规模数据上运行慢于排序实现的算法；
2. 在 50000000 个数的超大规模的数据集上，线性时间的算法快于排序实现的算法，验证了算法复杂度的“渐进性”。

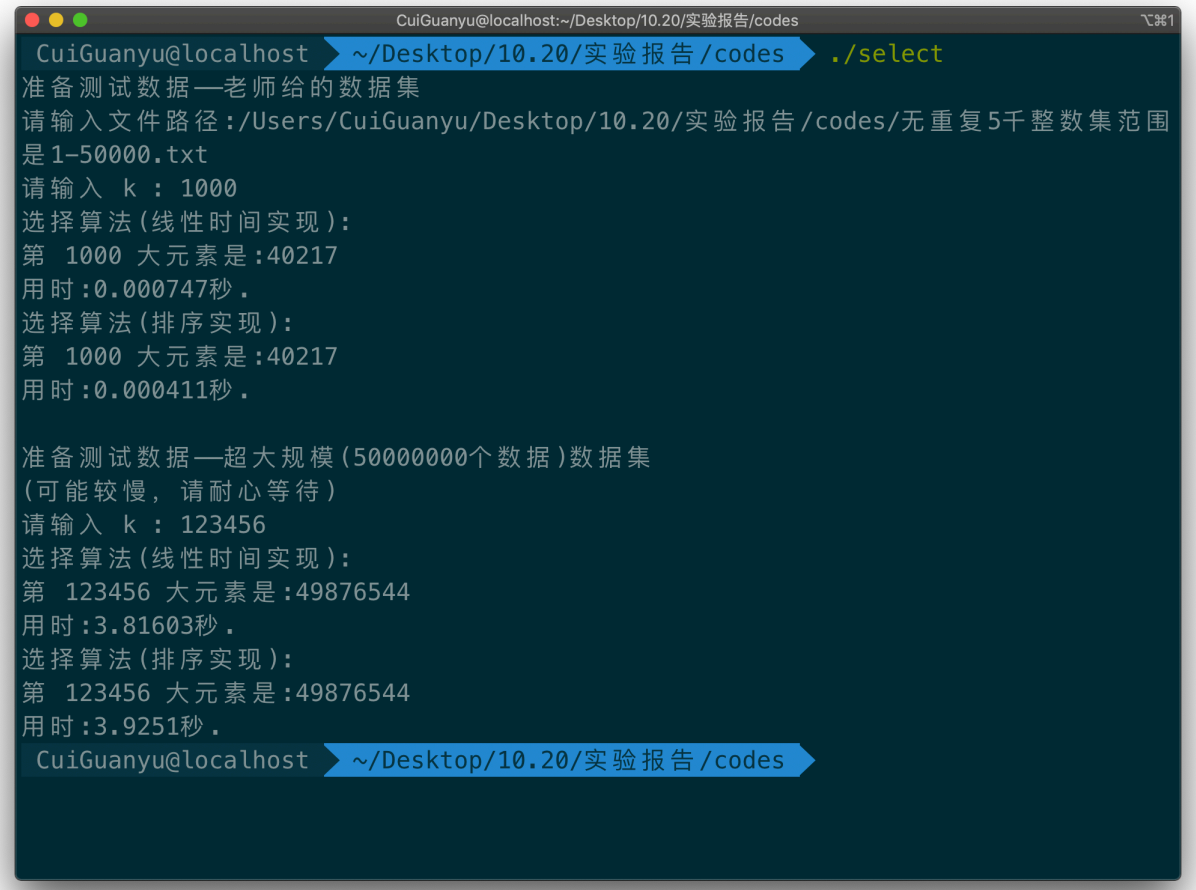
CuiGuanyu@localhost:~/Desktop/10.20/实验报告/codes
⌥⌘1
CuiGuanyu@localhost ~/Desktop/10.20/实验报告/codes ./select
准备测试数据——老师给的数据集
请输入文件路径:/Users/CuiGuanyu/Desktop/10.20/实验报告/codes/无重复5千整数集范围是1-50000.txt
请输入 k : 1000
选择算法(线性时间实现):
第 1000 大元素是:40217
用时:0.000747秒.
选择算法(排序实现):
第 1000 大元素是:40217
用时:0.000411秒.
准备测试数据——超大规模(50000000个数据)数据集
(可能较慢, 请耐心等待)
请输入 k : 123456
选择算法(线性时间实现):
第 123456 大元素是:49876544
用时:3.81603秒.
选择算法(排序实现):
第 123456 大元素是:49876544
用时:3.9251秒.
CuiGuanyu@localhost ~/Desktop/10.20/实验报告/codes

图 1: select 测试