HW7 中位数

赵楷越 522031910803

1 两种选择算法的实现选择

对于 quickSelect 方法和 linearSelect 方法，我均选择了参考书上的算法和伪代码进行了相应代码的编写。基于减而治之、逐步逼近的思路，可实现 quickSelect 算法。

```
template <typename T> void quickSelect ( Vector<T> & A, Rank k ) { //基于快速划分的k选取算法
   for ( Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
      Rank i = lo, j = hi; T pivot = A[lo];
      while ( i < j ) { //O(hi - lo + 1) = O(n)
         while ( ( i < j ) && ( pivot <= A[j] ) ) j--; A[i] = A[j];
         while ( ( i < j ) && ( A[i] <= pivot ) ) i++; A[j] = A[i];
      } //assert: i == j
      A[i] = pivot;
      if ( k <= i ) hi = i - 1;
      if ( i <= k ) lo = i + 1;
   } //A[k] is now a pivot
}
```

延续 quickSelect 算法的思路，以下 k-选取算法在最坏情况下运行时间依然为 O(n)。

```
select(A, k)
输入：规模为n的无序序列A，秩k ≥ 0
输出：A所对应有序序列中秩为k的元素
{
   0) if (n = |A| < Q) return trivialSelect(A, k); //递归基：序列规模不大时直接使用蛮力算法
   1) 将A均匀地划分为n/Q个子序列，各含Q个元素; //Q为一个不大的常数，其具体数值稍后给出
   2) 各子序列分别排序，计算中位数，并将这些中位数组成一个序列; //可采用任何排序算法，比如选择排序
   3) 通过递归调用select()，计算出中位数序列的中位数，记作M;
   4) 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）;
   5) if (|L| ≥ k) return select(L, k);
      else if (|L| + |E| ≥ k) return M;
      else return select(G, k - |L| - |E|);
}
```

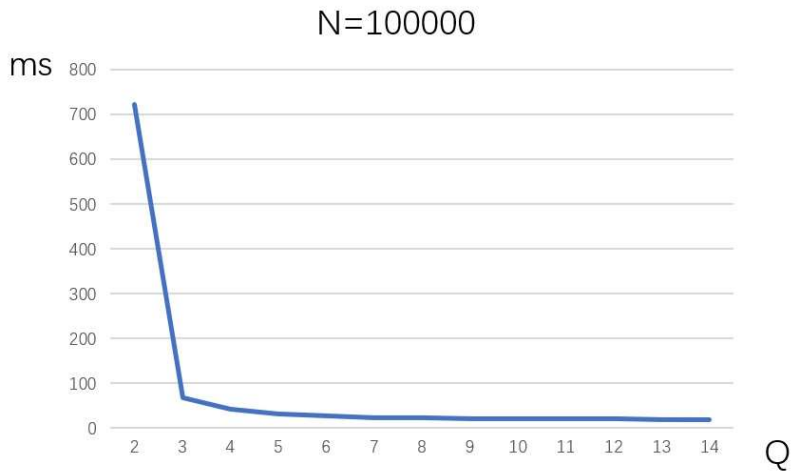
算法12.1 线性时间的k-选取

2 实验及对应结果分析

为了研究 Q 值对 linearSelect 效率的影响，我选择测试在数组大小 N 分别为 10000，50000，100000 的规模情况下，使数组为随机生成的数字，改变 Q 使 Q 取 2-14 中的一值时，linearSelect 算法的效率。对应实验结果表格如下所示。

Q值对Linear Select效率的影响													
N=10000													
Q	2	3	4	5	6	7	8	9	10	11	12	13	14
ms	58	6.5	4.6	3.3	2.9	2.5	2.4	2.3	2.1	2.1	2	2	2
N=50000													
Q	2	3	4	5	6	7	8	9	10	11	12	13	14
ms	350	32	22	15.7	13.9	12.3	11.6	10.8	10.6	10.3	10.5	10.3	10.4
N=100000													
Q	2	3	4	5	6	7	8	9	10	11	12	13	14
ms	722	67	43	31.5	27.5	24.3	22.7	22	21.1	20.8	20.3	20	19.8

将其中 N=100000 时的情况绘制成折线图。我们发现当 Q=9 时，linearSelect 算法便可以达到一个最优效率。而算法的效率随着 Q 的增大而提高。



为了将 linearSelect 与 quickSelect 算法进行对比，我们分别在随机数据、顺序数据、逆序数据的三种情况下，使数据规模 N 从 10000 递增至 100000，研究其算法效率与数据规模和数据特性（顺序或乱序的情况下）的关系。因为当 Q 为 9 时，linearSelect 的性能已经最优平稳，因此在实验中我们选择 Q=9。实验数据表格如下所示。

Linear Select与Quick Select对比										
随机数据										
Linear Select(Q=9)										
N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
ms	2.4	3.5	6.6	8.8	11.1	13.2	15.7	17.5	19.5	25.3
Quick Select										
N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
ms	0.17	0.55	0.68	0.91	1.33	1.45	1.65	1.66	2.22	2.78
顺序数据										
Linear Select(Q=9)										
N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
ms	1.2	4.8	6.6	5.9	7.7	8.5	10.2	11.4	11.9	7
Quick Select										
N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
ms	73	293	660	1179	1835	2661	3599	4709	5963	7375
逆序数据										
Linear Select(Q=9)										
N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
ms	2.1	4	6.2	7.9	9.9	11.8	13.9	15.6	17.5	19.8
Quick Select										
N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
ms	97	390	880	1574	2435	3557	4778	6229	7907	9727

我们发现，在数组数据有序的情况下，quickSelect 算法的性能极差，因为它每次选取了最低位上的数据作为 pivot，但数据又是有序的，因此导致算法性能崩溃。而 linearSelect 在不同规模，不同类型的数据情况下表现均良好。表明了其线性性能特征。下表是对在逆序数据情况下，linearSelect 和 quickSelect 算法的性能随数据规模的对比。