Report

CS214-Algorithm and Complexity, Spring 2018

Name: 杨培灏　Student ID: 516021910233

Problem: 介绍如何得到序列中前 k 小的数（不是第 k 小!）。报告中至少包括算法介绍、复杂性分析，也至少要包括一种在大数据情况（数据量超过内存）下的算法。

Defination: 序列 A，长度为 N，$swap(a,b)$ 方法表示交换两个参数的值，$max()$ 方法返回参数的最大值。

首先讨论允许重复数字出现的前 k 小的数，即 k 个数，而不一定是 k 种。

1. 利用直接插入排序，全部排递增序后输出前 k 个数。最好情况下时间复杂度为 $\mathcal{O}(N)$, 最坏情况下为 $\mathcal{O}(N^2)$。空间复杂度为 $O(1)$。

算法 1: straightInsert()

```
1 for i = 1 → N do
2 |   temp=A[i]; for j = i − 1 → 1) and A[j]<temp do
3 |   |   A[j+1]=A[j];
4 |   A[j]=temp;
5 output A[1],...,A[k];
```

2. 利用希尔排序，最好情况下时间复杂度为 $\mathcal{O}(N)$, 最坏情况下为 $\mathcal{O}(N^2)$。也是全部排递增序后输出前 k 个数。希尔排序大大减少了调整过程中移动的操作，但其实际运行效果与增量参数选择有很大关系。空间复杂度为 $\mathcal{O}(1)$。

算法 2: shellSort()

```
  Input: 增量 gap
1 while gap>0 do
2 |   for i = gap → N do
3 |   |   j=i;
4 |   |   while j-d>0 do
5 |   |   |   if A[j]<A[j-d] then
6 |   |   |   |   swap(A[j],A[j-d]);
7 |   |   |   j=j-d;
8 |   gap=gap/2;
9 output A[1],...,A[k];
```

3. 利用直接选择排序的思路，直接输出前 k 个最小数，时间复杂度为 $\mathcal{O}(kN)$，空间复杂度为 $\mathcal{O}(1)$。

算法 3: straightSelect()

```
1  minimum=A[1];
2  pos=1;
3  //k 次直接选择输出序列中最小的数。
4  for i = 1 → k do
5      For(j in (i,…,N)) if A[j]<minimum then
6          minimum=A[j];
7          pos=j;
8      output minimum;
9      A[pos]=A[i];
10     minimum=A[i+1];
11     pos=1;
12 output A[1],…,A[k];
```

4. 利用堆排序的思路建最小堆。$\mathcal{O}(N)$ 时间建堆，之后每次调整可以直接输出最小元，$\mathcal{O}(k \log N)$ 时间排序输出，故总的时间复杂度为 $\mathcal{O}(k \log N + N)$。在序列的基础上直接就地建堆的话空间复杂度为 $\mathcal{O}(1)$。采用自底向上建小堆

算法 4: down2upBuildHeap()

```
1  for i = ⌊N/2⌋ → 1 do
2      while 2i + 1 ≤ N do
3          if A[2i]<A[2i+1] then
4              if A[i]>A[2i] then
5                  swap(A[i],A[2i]);
6                  i=2i;
7          else
8              if A[i]>A[2i+1] then
9                  swap(A[i],A[2i+1]);
10                 i=2i+1;
11     if 2i=N then
12         if A[i]>A[2i] then
13             swap(A[i],A[2i]);
```

算法 5: percolateDown()

```
1  //就是上面自底向上建堆的方法 temp=A[1];
2  now=1;
3  while 2 ∗ now ≤ N do
4      child=2*now;
5      if child<N and A[child]<A[child+1] then
6          child=child+1;
7      Iftemp>A[child] A[0]=A[child];
8      now=child;
9  A[now]=temp;
```

算法 6: solve()

```
down2upBuildHeap();
for i = 1 → k do
    output A[1];//输出最小堆的一号元素即最小元
    swap(A[1],A[N+1-i]);
    percolateDown();
```

5. 利用冒泡排序对整个序列进行排序，输出前 k 个数。最好情况下时间复杂度为 $\mathcal{O}(N)$，最坏情况下时间复杂度为 $\mathcal{O}(N^2)$。其空间复杂度为 $\mathcal{O}(1)$。

算法 7: bubbleSort()

```
flag=true;
while flag do
    flag=false;
    for i in (2,N) do
        if A[i-1]>A[i] then
            swap(A[i-1],A[i]);
            flag=true;
output A[1],…,A[k];
```

6. 利用快速排序的思想，但是这里我们并不需要完整的序，所以修改原始算法为：如果标记元素左侧小于它的元素数量已经超过了 k，就不需要对右边元素排序了。在排递增序后，输出前 k 个数。时间复杂度略微小于初始的 $\mathcal{O}(N \log N)$。

算法 8: quickSort(low,high)

```
Input: 排序的起点和终点 low、high
if low ≥ high then
    return;
first=low;
last=high;
key=A[first]; while last>first do
    while A[last]>key and last>first do
        last=last-1;
    A[first]=A[last];
    while A[first]<key and last>first do
        first=first+1;
    A[last]=A[first];
A[frist]=key;
quickSort(low,first);
if first-low<k then
    quickSort(first+1,high);
```

7. 利用归并排序，时间复杂度为 $O(N\log N)$，一般地有 $T(n)=2T(n/2)+O(n)$。但是子问题排序成递增序后，最后一步比较特殊，只需要合并出前 k 个就行了，$T(N)=2T(N/2)+O(k)$。故总的复杂度为 $\mathcal{O}(N\log(N/2)+k)$，但是其空间复杂度会到 $\mathcal{O}(N)$。

算法 9: mergeSort(start,end)

```
Input: 排序起点和终点 start、end
temp 为一个长度为 N 的辅助序列。
if start ≤ end then
    return;
else
    mid=(start+end)/2;
    mergeSort(start,mid);
    mergeSort(mid+1,end);
    i=start; j=mid+1; h=start;
    if end-start=N-1 then
        while i<mid-start+1 and j<end-mid and h!=k do
            if A[i]<A[j] then
                temp[h]=A[i];i=i+1;h=h+1;
            else
                temp[h]=A[j];j=j+1;h=h+1;
        while i<mid-start+1 and h!=k do
            temp[h]=A[i];i=i+1;h=h+1;
        while j<end-mid and h!=k do
            temp[h]=A[j];j=j+1;h=h+1;
        for h = 1 → k do
            A[h]=temp[h];
    else
        while i<mid-start+1 and j<end-mid do
            if A[i]<A[j] then
                temp[h]=A[i];i=i+1;h=h+1;
            else
                temp[h]=A[j];j=j+1;h=h+1;
        while i<mid-start+1 do
            temp[h]=A[i];i=i+1;h=h+1;
        while j<end-mid do
            temp[h]=A[j];j=j+1;h=h+1;
        for h = start → end do
            A[h]=temp[h];
```

8. 我们还可以将数据分割开处理。

   这里假设 $N\gg k$，那么将数据按存储时的连续顺序分为 N/k 组（记作 n），对每组数据，采用堆排序或快速排序，那么可以得到 n 组递增序子序列，这一操作的时间复杂度为 $N/k\times k\log k$，即 $\mathcal{O}(N\log k)$。

再将这 n 组中的最小元进行类似的排序，这一步的时间复杂度为 $\mathcal{O}(n\log n)$。事实上，总体的最小元，只可能出现在按最小元大小排递增序后的前 k 组子序列中。

那么接下来，我们只需要从 k^2 个元素中挑选即可，最好情况下我们只需要寻找 k 次，即第一组中的每个元素均比第二组的最小元小。最坏情况下我们需要检查每个元素。

整个算法的时间复杂度为 $\mathcal{O}(max(N\log k, n\log n))$。

9. 接下来讨论几种空间复杂度较高的方法，他们不借助元素间的比较。这里我们假设序列中均为正数（若序列中存在负数，可以均加上一个数值作为计数时的偏移量，不影响大小关系），则可以借用计数排序思路，读取序列，统计每个值出现的次数，最后再输出前 k 小的数。时间复杂度为 $\mathcal{O}(N)$，遍历一次即可。缺点是辅助空间开销大，为 $\mathcal{O}(max(A[1],\ldots,A[N]))$。

算法 10: countSort()

```
Input: 辅助数列 M，长度为 max(A[1],...,A[N])，元素初始为 0
for i = N → 1 do
    M[A[i]]=M[A[i]]+1;
j=k;
while j > 0 do
    if M[i]>0 then
        output i;
        j=j-1;
        M[i]=M[i]-1;
    else
        i=i+1;
```

10. 同样假设序列中均为正数，那么借用桶排序的思路，可以将数据统一除以一个值 gap，比如 10，那么可以将数据存入不同的桶中。事实上，不包含第 k 小的元素的桶内的数据顺序是无所谓的，我们只需要从小的桶开始直接输出，当发现输出的数字数量大于 k 时，收回当前桶的输出，然后对该桶内数据排序，重新输出到 k 个数就停止即可。时间复杂度同样为 $\mathcal{O}(N)$。但是与上面的计数方法一样，会有较高的空间复杂度。

算法 11: bucketSort()

Input: 存储 A 中元素的 bucket，桶内数据的跨度 gap，bucket 的数列 buckets，长度为 $max(A[1],\dots,A[N])/gap$

```
for i = N → 1 do
    buckets[A[i]/gap].push(A[i]/gap);
j=k;
while j>0 do
    if buckets[i]>0 then
        if j=j-buckets[i].size()>0 then
            output buckets[i];//输出该桶内所有元素
            j=j-buckets[i].size();//减去该桶内元素数量
        else
            bucket[i].sort();//对桶内元素进行排序
            while j>0 do
                顺次输出桶内元素;j=j-1;
            break;
    i=i+1;
```

11. 借用基数排序的思想，我们只采用十个桶，分别代表一位上的数字 0 到 9，根据 A[i] 第 p 位的大小 $(A[i]\%10^{p+1}/10^{p})$ 分配到这些桶中。

    由于我们并不需要完整的序，我们可以从一个很大的位数开始逐渐抛弃元素到一个合理的大小再进行基数排序。比如取 10^5，如果零号桶内元素数量大于 k，那么后面的桶内元素均可以丢弃。

    对剩余较少的元素采用基数排序，先由个位分类到十个桶中，然后收集起来，再按十位分类，收集……直到完全增序，输出前 k 个。一般地，该方法的时间复杂度在 $\mathcal{O}(rN)$，r 为重复的趟数，但是我们首先排除了大量数据，因此可以认为这是一个 O(N) 的方法，同时它可以完全避免数据间的比较。

下面讨论数据量极大的情况。sort() 表示对内存内的数据进行排序处理，排序方法为任意方法。

1. 对于数据量巨大的情况，考虑到空间代价，我们无法直接使用 7、9、10、11 中的方法。由于我们需要在内存外存间进行数据交换，而这个过程是非常缓慢的，所以我们的目标在于如何减少外存的读写。最常用的外排序是利用了归并排序的思想。整个过程分为预处理和归并两个阶段。

   预处理最简单的方法是按照内存的容量尽可能多地读入数据记录，然后在内存进行排序，排序的结果写入文件，形成一个已排序片段。已排序片段内记录越多，归并的时间就会缩短，置换选择可以让我们在只能容纳 p 个记录的内存中生成平均长度为 2p 的初始的已排序片段。该方法如下：

   先将内存中的元素建堆 Q 为优先级队列，那么 Q.deque() 输出一个元素，可以从外存读入另一个元素。如果它比输出元素大，那么它可以加入当前排序片段，那么它可以入队，在堆中进行调整；否则它无法加入当前排序片段，暂存在内存，但是不入队。持续这个过程，直到队列为空。此时一个已排序片段形成。

   之后类似归并排序的方法，利用二路归并方法将这些已排序片段归并。采用 m 路归并 p 个数据段时，访问外存的次数为 $\log_m p$，显然多路归并较二路归并在外存的访问次数上有着优势。但是在归并时，如果依次比较，则比较次数为 $c=m-1$，那么总的比较次数为 $\lfloor\log_m p\rfloor\times c$，

即 $\lfloor \log m / \log p \rfloor \times c$，这里我们可以利用败者树，降低多路归并时的比较次数，优化 $c = \log k$，这时总的比较次数为 $\lfloor \log m \rfloor$。

败者树是一个二叉树，该树其实很类似堆，只是额外保留了叶子节点存储所有数据，用父亲节点存储失败者。该二叉树的建立过程就类似自底向上建堆：输入每个归并段的第一个记录作为归并树的叶子节点。儿子节点两两相比较，父亲节点存储了两个节点比较的败者（较大的值）；胜利者（较小者）可以参与更高层的比赛。这样树的顶端就是当次比较的亚军（次小者），相应的，冠军（最小者）被输出。当我们把败者树中最小者输入到输出文件以后，需要从被输出元素（之前的冠军）对应的归并段取出下一个记录补回败者树中，再从树中选出一个新的最小元。这个过程我们就叫做调整败者树，我们只需要沿着当前节点的父亲节点一直比较到顶端。比较的规则是与父亲节点比较，比较小的（胜者）可以参与更高层的比较，即可以跟它爷爷比较，一直向上，直到根节点，过程中大的（失败者）留在当前节点。显然，这个过程又类似于最小堆的调整。

同时应该注意到我们不要求整体有序，事实上，这里类似普通的归并排序，在最后的归并操作可以进行优化，只进行到排出最小的 k 个元素即可。

下面是利用败者树进行多路归并的算法：

算法 12: treeMerge()

```
Input: m 路归并，读入的文件数据 file，各路文件读取进度 pointer
while do
    input file;
    for i = 1 → m do
        pointer[i]=1;//各路 file 均从第一个数据开始读取
        tree[m+i]=file[i][pointer[i]];
        //将读入的 m 路 file 的第一个数据存到树的叶子节点中
        winner[m+i]=tree[m+i];
        //叶子层也可以看作胜利者，因为根据它们向上建立父亲节点。这样写方便下面的循环里书写
    for i = m → 1 do
        tree[i]=max(winner[2i],winner[2i-1]);
        //败者树父亲节点存储的是儿子节点中的失败者、较大值
        winner[i]=min(winner[2i],winner[2i-1]);
        //胜利者更新，下次根据新的胜利者、较小值建立父亲
    while file 未读取完毕 do
        output winner[1];//输出当前冠军，并且假设 winner[1] 来自第 t 路 file
        pointer[t]=pointer[t]+1;
        tree[m+t]= 读入 file[t][pointer[t]];
        winner[m+t]=tree[m+t];
        for i = m → 1 do
            //调整败者树和胜利者数组
            tree[i]=max(winner[2i],winner[2i-1]);
            winner[i]=min(winner[2i],winner[2i-1]);
```

2. 在面对海量数据的时候，还可以分而治之。注意到我们在第 8 个方法中拆分了数据，由于实际上外存的读取不是上面外排序描述的单个元素的读取，而是一次从外存读取成块的大量数据，所以方案 8 可能更适宜，同时我们采用方案 8 可以舍弃大量的数据，从而最终直接在内存排序。所以我认为第 8 个方案可以运用在大量数据的处理中。如果外存读取速度较快，我

认为方案 11 中的基数排序的思路也可以考虑，先遍历一次，抛弃掉过大的元素，剩余元素可以放在内存中讨论。

3. 也可以不使用比较的方法，类似之前的桶排序思路，但是这次建立二叉搜索树进行统计，当数据超过 k 种时，可以对二叉树进行剪枝，这样维护的永远是最大的 k 个数的出现次数，这样内存中的空间可以被节约下来，只需要不断读入数据即可，这样遍历一次就可以得到前 k 小的数。

算法 13: binaryTree()

```
Input: 读取部分输入文件 file
build tree; while 未读取完外部文件 do
    Input file;
    for item in file do
        if tree.has(item) then
            tree[item]=tree[item]+1;//树中包含该数据，那么该数记录加 1
        else
            tree.insert(item);//数据大的为右儿子，小的为左儿子
        maintain();//维护树的规模，保证树的结点数不大于 k;
遍历二叉搜索树，输出这 k 个最小的数;
```

最后讨论一个特殊的情况：如果题目中要求的前 k 小的数，是不允许重复数字出现的前 k 小的数，即 k 种数，而不一定是 k 个。采用上方各种方法，可以增加额外的变量记录数字的种类即可。当然还有另一种思路是位图方案，其实很类似之前提到的桶排序方法。只不过由于这里考虑到了不允许重复出现的特性，可以用更小的空间去存储，比如这里用到的比特串，所以针对大数据也可以适用。下面是伪代码

算法 14: bitBucket()

```
Input: 读取部分输入文件 file
for i = 1 → N do
    bit[i]=0;//各位置 0
//读入文件中的整数，将比特串的对应位置更改为 1。
for item in file do
    bit[item]=1;
for i = N → 1 do
    if bit[i] then
        output i;
        k=k-1;
    if k=0 then
        break;
//检验每一位，如果该位为 1，就输出对应的整数，直到输出了 k 个数
```