在序列中寻找前 k 小数

刘欣鹏 516030910259

1. 问题描述

现有一个长为 n 的任意序列，给出寻找该序列中前 k 小数的算法。

2. 算法整理

2.1. 基于选择排序的算法

2.1.1. 基本思路

初始答案集合 S 置为空。对序列进行 k 次扫描，在第 k 次扫描中寻找序列中剩余的 n-k+1 个数中最小的数，将其从序列中删除并加入答案集合。扫描结束后，答案集合 S 即为所求。

2.1.2. 伪代码

输入：初始数组 a[1..n]；k。

```
For i = 1 to n do
begin
    For j=i+1 to n do
        If (a[j]<a[i]) swap(a[i],a[j]);————(a)
 End
 A[1..k]即为所求。
```

2.1.3. 时间复杂度分析

该算法的主要部分为(a)。该部分共执行 $\sum_{i=1}^{k} n-i+1=\frac{k(2n-k+1)}{2}=O(nk)$ 次。故该算法为一个 O(nk)算法。

2.2. 基于排序的算法

2.2.1. 基本思路

对序列直接快速排序，得到升序序列。新序列中前 k 个数即为所求。

2.2.2. 伪代码

```
Void sort(a,l,r){
    Int i=l, j=r, mid=a[(l+r)>>1], tmp;
    While (true){
        While (a[i]<mid) i++;
        While (a[j]>mid) j--;
        If (i<=j){tmp=a[i];a[i]=a[j];a[j]=tmp;i++;j--;}
        If (i>j) break;
    }
    If (i<r) sort(a,i,r);
    If (l<j) sort(a,l,j);
}
输入：初始序列 a[1..n]；k；
Sort(a,1,n);
a[1..k]即为所求。
```

2.2.3. 时间复杂度分析

该算法时间复杂度即为快速排序的时间复杂度：O(nlogn)。

2.3. 基于堆的 O(klogn)算法

2.3.1. 基本思想

对序列 a[1..n]建立一个小根堆。置答案集合 S 为空。对以下操作，执行 k 次：删除当前堆顶并将其置入 S；维护堆。此算法是对 2.1 算法的堆优化。

2.3.2. 伪代码

```
输入：a[1..n]；k
//建堆开始
For i=n downto 1 do{
      j=i;
      While (j/2){
            If (a[j]<a[j/2]) {swap(a[j],a[j/2]);j/=2;}
            Else break;
      }
}
//建堆结束
For i=n downto n-k+1 do{
      swap(a[i],a[1]);
      j=1;
      While (j*2<=i-1) do{
            if (a[j*2]<a[j*2+1]) p=j*2;
            Else p=j*2+1;
            If (a[p]<a[j]) {swap(a[p],a[j]);j=p;}
            Else break;
      }
}
a[n..n-k+1]即为所求。
```

2.3.3. 时间复杂度分析

建堆部分，采用自底向上建堆方法，时间复杂度为 O(n)。

算法主要部分，共进行 k 次调整堆，每次最坏需进行 O(logn)次操作，时间复杂度为 O(klogn)。

故算法时间复杂度为 O(n+klogn)=O(klogn)。

2.4. 基于堆的 O(nlogk)算法

2.4.1. 基本思想

维护一个大小为 k 的大根堆。扫描 a[1..n]。对 a[i]，若 a[i]不小于堆顶元素，不做任何操作；若 a[i]小于堆顶元素，删除堆顶元素，将 a[i]加入堆中。最终，堆内元素即为所求。

2.4.2. 伪代码

```
输入：a[1..n]；k。
\\建堆
For i=k downto 1 do{
      j=i;
      While (j/2){
            If (a[j]<a[j/2]) {swap(a[j],a[j/2]);j/=2;}
            Else break;
```

```
        }
    }
    \\结束建堆
    For i=k+1 to n do{
        If (a[i]<a[1]) {
            a[1]=a[i];
            j=1;
            While (j*2<=k){
                If (a[j*2]>a[j*2+1]) k=j*2;
                Else k=j*2+1;
                If (a[j]<a[k]) {swap(a[j],a[k]);k=j;}
                Else break;
            }
        }
    }
```

a[1..k]即为所求。

2.4.3. 时间复杂度分析

建堆部分，采用自底向上方法，时间复杂度为 O(k)。

主要部分，最坏情况需进行 n-k+1 次堆的调整，每次为 O(logk)，共为 O(nlogk)。

故总代价为 O(nlogk)。

2.5. 基于哈希的算法

2.5.1. 基本思想

对数据进行不破坏大小顺序的哈希，记录每个哈希值出现的次数。最后从小到大扫描哈希表并计数。计数达到 k 时，此前已被扫描到的元素即为所求。

2.5.2. 伪代码

```
输入：a[1..n], k;
辅助变量:h[1..hash(n)];\\哈希表
For i=1 to n do    h[hash(a[i])]++;
ans={};
Count=0;
For i=h.begin to h.end do
begin
    Count+=h[i];
    ans.insert(hash^-1(i));
    If (Count>=k) break;
End
```

ans 即为所求。

2.5.3. 时间复杂度分析

在寻找到适当的哈希函数时，初始数组内每个元素被扫描一次，哈希表内元素最多访问 k 个，故总时间代价为 O(n+k)。

2.6. 基于快速排序的算法

2.6.1. 基本思想

采取快速排序“取中值后分割的思想”，主体函数为 find_k(l,r,p)。l 为当前序列左端点，r 为当前序列右端点，p 意为要在 a[l..r]内寻找前 p 小的数。

具体实现见伪代码。

2.6.2. 伪代码

输入：a[1..n]；k

```
void find_k(int l,int r,int p){
    Int i=l, j=r, mid=a[(l+r)/2];
    While (true){
        While (a[i]<mid) i++;
        While (a[j]>mid) j--;
        If (i<=j) {swap(a[i],a[j]);i++;j--;}
        If (i>j) break;
    }//分割 a[l..r]为两部分
    if (j-l=p) return;//若左半部分大小为 p，则已找到 a[l..r]内前 p 小的数
    Else if (j-l>p) find_k(l,j,p);//若左半部分大小大于 p，则需在左半部分继续查找
    Else if (j-l<k) adjust_buffer(i,r,p-l+j);//若左半部分大小小于 p，则除了保留左半部分，亦需在右半部分查找
}
find_k(1,n,k);
Qsort(1,k);//对前 k 小数进行快速排序。
```

a[1..k]即为所求。

2.6.3. 时间复杂度分析

find_k 函数部分，期望执行时间为 $\sum_{i=1}^{\log n}\frac{n}{2^i}=2n=O(n)$。

快速排序部分时间代价为 O(klogk)。

故总时间代价为 O(n+klogk)。

2.7. 针对大数据的算法 1

2.7.1. 适用条件

n 大于内存空间范围，k 小于内存空间范围。

2.7.2. 基本思想

与 2.4 中描述的算法基本一致。由于 k 小于内存空间范围，我们可以在内存内维护一个大小为 k 的大根堆，每次从内存中读取新的元素与堆顶比较。若小于堆顶元素，则替换堆顶元素并重整堆；否则不进行操作。

2.7.3. 伪代码

与 2.4 算法伪代码基本一致。

2.7.4. 时间复杂度分析

由于 n 大于内存空间范围，在此情况下，我们必须引入外存。而对外存的读取操作花费的时间远超对内存中元素的操作，故本算法的时间复杂度主要是对元素的读取，时间复杂度为 O(n)。对内存中元素的操作最坏情况下时间复杂度为 O(nlogk)，在 2.4 中已作说明。

2.8. 针对大数据的算法 2

2.8.1. 适用条件

k 小于内存空间范围的一半。

2.8.2. 基本思想

建立一个大小为 2k 的缓冲区，每次向缓冲区中读入数据。当缓冲区满后，用快速排序找中值分割的思想取出缓冲区中前 k 小的数并删去其他的，直到所有数据都被读入过了。最终缓冲区

内的前 k 小的数据即为所求。

2.8.3. 伪代码

```
load(k);//初始化，读入 k 个数据
While (未读完数据){
    load(k);\\读入 k 个数据
    Adjust_buffer(1,2k,k);\\调整缓冲区；函数实现见下
    删除 buffer[k+1..2k];
}
Sort(1,2k);\\快速排序
buffer[1..k]即为所求。

void adjust_buffer(int l,int r,int p){
    Int i=l, j=r, mid=buffer[(l+r)/2];
    While (true){
        While (buffer[i]<mid) i++;
        While (buffer[j]>mid) j--;
        If (i<=j) {swap(buffer[i],buffer[j]);i++;j--;}
        If (i>j) break;
    }
    if (j-l=p) return;
    If (j-l>p) adjust_buffer(l,j,p);
    If (j-l<k) adjust_buffer(i,r,p-l+j);
}\\与 2.6 中算法思想基本一致
```

2.8.4. 时间复杂度分析

每个数据需访问一次，代价为 O(n)；调整缓冲区部分每次期望代价为 O(k)（2.6），总代价期望为 O(n)；最终对缓冲区内元素进行快速排序，代价为 O(klogk)。

所以，算法总时间代价为 O(n+klogk)。当 k<<n 时，可近似认为总时间复杂度为 O(n)。

2.9. 针对大数据的 $O(\frac{k}{m-1}n\log(\frac{k}{m-1}))$ 算法（m 为内存空间范围）

2.9.1. 适用条件

n,k 均大于内存空间范围。

2.9.2. 基本思想

该算法仍基于 2.4 中描述的算法。由于 k>m，我们可以分 $\frac{k}{m-1}$ 次求解，第 i 次求解第 $\frac{(i-1)k}{m-1}$ 小到第 $\frac{ik}{m-1}$ 小的数。

每次求解时，首先建立一个大小为 m-1 的大根堆，剩余的一个空间单位存储上一次求解中得到的最大元素 x。然后扫描 a[1..n]，若当前元素大于 x 且小于堆顶，则用其替换堆顶并重新调整堆；否则不进行操作。

求解完成后，用堆顶元素替换 x 并将堆内所有元素存入外存，进行下一次求解，直到结束。所有被存入外存的元素即为所求。

2.9.3. 伪代码

```
Buildheap(m-1);//自底向上建堆
For i=1 to n do{
```

```
            If (heap[1]<a[i]<x) {
                swap(heap[1],a[i]);//用 a[i]替换堆顶
                adjust(heap,1);//维护堆的性质
            }
        }
        x=heap[1];
        Save(heap);//将堆内元素存入外存。
```

2.9.4. 时间复杂度分析

对于给定的 n，k 和 m，共需进行 $\frac{k}{m-1}$ 次求解，每次求解的时间复杂度为 $O(n\log(\frac{k}{m-1}))$ 。故总时间复杂度为 $O(\frac{nk}{m-1}\log(\frac{k}{m-1}))$ 。

3. 分析与比较

在以上罗列的八种算法中，前六种算法针对较小数据规模，最后三种针对大数据规模。

在较小数据规模中，发现时间复杂度方面表现最好的是基于哈希的 O(n)算法，但由于可能难以找到合适的哈希函数以避免冲突问题，实用性并不好。而其他算法均是基于比较的算法。因此，我们不妨借助决策树证明该问题的下界。

该问题的输入是序列 x1,x2,...,xn，输出是排序后的前 k 小数。把输出看做是输入的一个排列 A_k^n，则每一种排列都是一个可能的输出。若算法对于所有可能的输入都能处理，则该算法是正确的。故决策树共有 A_k^n 种可能输出，必须被不同的叶子节点表示。所以，决策树的高度至少为 $\log(A_k^n)$ 。故算法的时间复杂度下界为 $O(k\log k)$ 之间。在我给出的四种算法中，2.6 最接近这一复杂度，在小数据情况下表现更优；而 2.3、2.4 与这一复杂度也比较接近，实现上也更容易理解。

在较大数据规模的情况下，我给出了三种算法，分别针对不同数据情况。相比之下，表现最优的是 2.7，复杂度近似为 O(n)，但适用的数据范围也最小；另外两种则均基于堆这一数据结构，虽然时间复杂度不是最好的，但适用范围更广。