学号：20184484
班级：计算机 1803
姓名：胥卜凡

目录

1 对于一个二进制流，设计空间复杂度为 O(K)的算法，计算长度为 N 的滑动窗口中 1 的个数（1<=K<=N）。

A)算法 1：考虑依照水库抽样算法进行算法设计

1.1 伪代码：

算法 1 计算滑动窗口中1的个数

输入： 目标长度k,数据流$A[1...N]$

输出： 滑动窗口中1的近似个数result

$B[1...k] = 0$;
$result$ =0;
y =0;
for i=1 **to** k **do**
    B[i]=A[i];
end for;
for i=k+1 **to** N **do**
    A[i]以$\frac{k}{i}$的概率随机替换B[1..k]中的一个元素
end for;
for i=1 **to** k **do**
    if $B[i]$==1 **then**;
        y ++;
    end if;
end for;
$result$ =$\frac{y \times N}{k}$
return $result$

1.2 分析：

（1） 算法设计：

①总体思想对长度为 N 的滑动窗口中的二进制流进行 k 个均匀抽样，然后将 1 所占的比例乘以长度 N 得到结果。

②过程：

（1）申请一个长度为 k 的数组 A 保存抽样。（2）保存首先接收到的 k 个元素。（3）当接收到第 i 个元素时，以 k/i 的概率随机替换 A 中的元素。即随机生成[1,i]间的数 j，当 j ≤k 时，替换 A[j]。（4）计算出数组 A 中 1 所占的比例 rate，使用 rate 与 N 相乘得到最后的结果）。

（2） 复杂度分析：

空间复杂度：内存中存的 K 个数，可以看做是 O（k）

时间复杂度：对数据流，显然是 O（N）

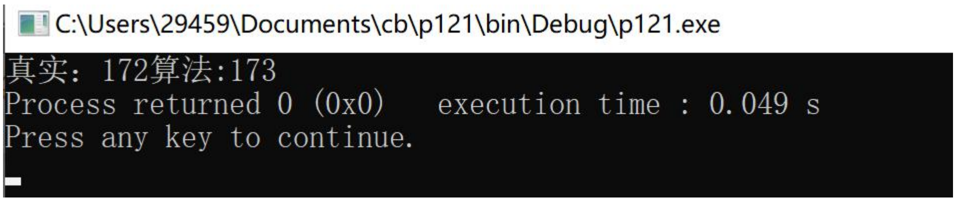
1.3 样例程序：

①代码：

```
#include  <iostream>
#include  <string>
#include  <ctime>
using  namespace  std;
int  main()
{
        string  A="0110011001001111110101001000000010101010101001011111111111111111111111111111111111111111110000000000000000000000000000000000000000000001111111111111111111111111110001010011110101010010101010101010010101010100101010101001010101010101010101011111111111111111111111111111111011010010111010101011010100101010101001";
//计算真实结果
        int  rel_1=0;
        int  len=A.size();
        for(auto  &i:A)
        {
                if(i==’1’)
                        rel_1++;
        }
        int  result=0;
        //算法执行过程
        for(int  j=0;  j<1000;  j++)
        {
                const  int  k=5;
                int  B[k]=  {};
                int  y=0;
                for(int  i=0;  i<k;  i++)
                {
                        B[i]=A[i];
                }
                unsigned  seed;
                seed=time(0);
                for(int  i=k;  i<len;  i++)
                {
                        float  pr=(k*1.0)/i;
                        int  r=rand()%100;
                        if(r<pr*100)
                        {
                                int  f=rand()%k;//随机替换的元素
                                B[f]=A[i];
                        }
```

```

            }
            for(int  i=0;  i<k;  i++)
            {
                if(B[i]=='1')
                    y++;
            }
            result+=y*len/k;
        }
        result=  result/1000;
        cout<<"真实："<<rel_1<<"算法:"<<result;
}
```

②结果：



由结果可知，小数据量的情况下算法的正确率还可以。

B)算法 2：考虑 DGIM 算法进行设计。

1.4 分析步骤

①思路：首先这个算法是用来回答这样的问题的“ 假设对于一个二进制流，我们有一个针对长度为 N 的滑动窗口（window）的查询“最后的 k 个位中有多少个 1 ”

因为数据流过大，无法存储全部数据，所以通过这个算法来进行大数据流统计。

算法步骤：

1）维护一个数据结构：将二进制分组，每组中 1 的个数是 2 的次幂，从右至左组的都是非递减的，只须记录每组的两端的位置即可。类似于 等比数列

2）这样组的个数有 O(log(2, N))，记录组的一端位置，所需空间 log(2, N)位。则统计整个窗口所占内存为 O((log(2, N)) ^ 2)

3）求解：估计 1 的个数：比较每组两端的位置与 k 的大小，找到 k 值所在的组 b，累加之前组的大小及组 b 的一半大小。

数据结构的更新：

1. 每一个新的位进入滑动窗口后，最左边一个位从窗口中移出（同时从桶中移出）；如果最左边的桶的时间戳是当前时间戳减去 N（也就是说桶里已经没有处于窗口中的位），则放弃这个桶；
2. 对于新加入的位，如果其为 0，则无操作；否则建立一个包含新加入位的大小为 1 的桶；
3. 由于新增一个大小为 1 的桶而出现 3 个桶大小为 1，则合并最左边的两个桶为一个大小为 2 的桶；合并之后可能出现 3 个大小为 2 的桶，则合并最左边两个大小为 2 的桶得到一个大小为 4 的桶……依次类推直到到达最左边的桶。

误差上下限：

- 至少是正确值的 50%：最差情况 b 中都是 1。
- 最多超过正确值的 50%：最差情况只有最右边一位为 1。

②复杂度

可以给出 DGIM 的存储开销：桶的个数 O(logN)，每个桶可以用 O(logN)空间表示，保存表示一个窗口的所有桶所占的空间为 O(log2N)。

对于查询，首先寻找含最近 k 个位的拥有最大时间戳的桶 b，查询结果为在 k 个为中所有桶的大小，加上 b 的大小的一半。一次查询的时间复杂度为 O(logN)。

③误差[5]：

假设最早的那个 bucket（记作 b_e）的 size 为 。因为在查询结果时我们加上了 b_e 的一半 size，当 b_e 只有结束位位于滑动窗口中时，此时会造成最大的误差，大小为。对于每个 size，滑动窗口中都至少存在一个 bucket，所以滑动窗口中 1 的真正个数不小于 。所以误差率最多为 ， 约等于 50%。[]

1.5 样例程序

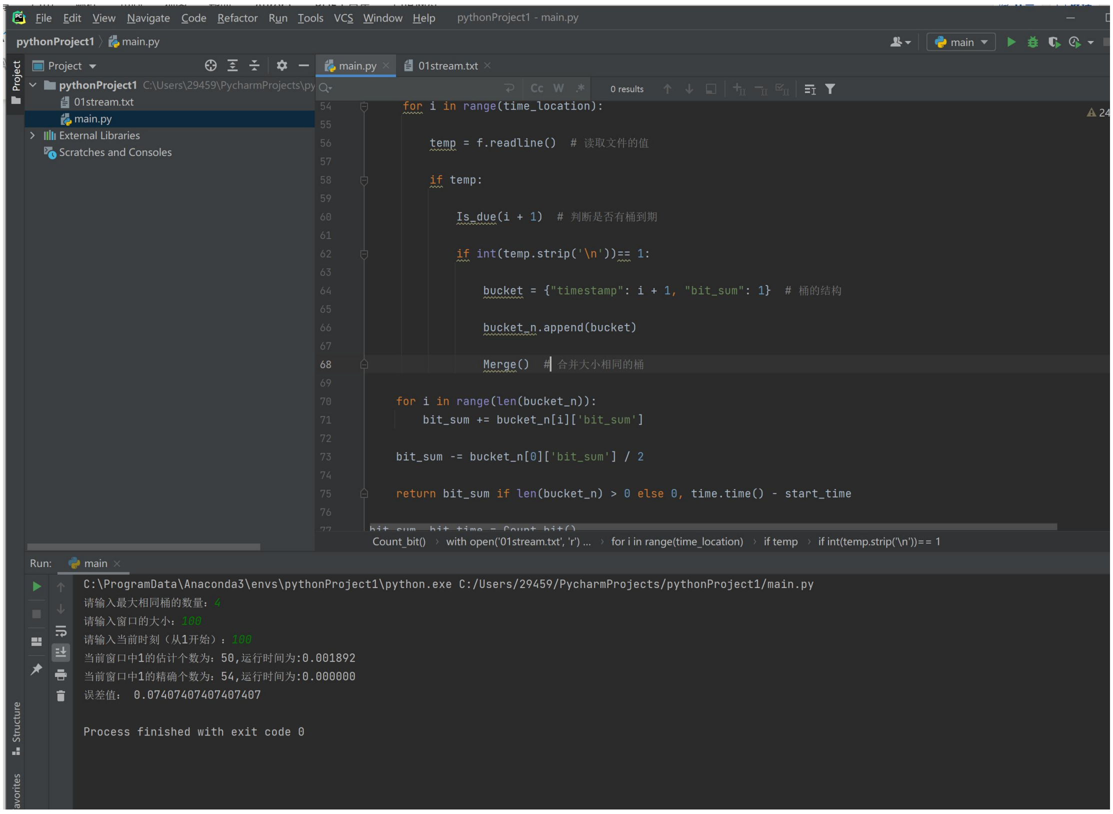
这里参考王老师的代码验证。

```
import time

bucket_n = []  # 桶的列表

n_max_bucket = int(input("请输入最大相同桶的数量："))

size_window = int(input("请输入窗口的大小："))

time_location = int(input("请输入当前时刻（从 1 开始）："))


```

```python
def Count_bit_act():
    bit_sum = 0  # 统计 1-bit 个数

    start_time = time.time()

    with open('01stream.txt', 'r') as f:
        f.seek(0 if time_location <= size_window else 2 * (time_location - size_window))  # 跳转到窗口大小之前行位置
        for i in range(time_location if time_location <= size_window else size_window):
            temp = f.readline()  # 读取值
            if temp and int(temp.strip('\n')) == 1:
                bit_sum += 1
    return bit_sum, time.time() - start_time


def Is_due(time_now):
    if len(bucket_n) > 0 and time_now - size_window == bucket_n[0]['timestamp']:  # 最左边的桶的时间戳等于当前时间减去窗口大小，到期了
        del bucket_n[0]


def Merge():
    for i in range(len(bucket_n) - 1, n_max_bucket - 1, -1):

        if bucket_n[i]['bit_sum'] == bucket_n[i - n_max_bucket]['bit_sum']:

          # 存在 n_max_bucket 个大小相同的桶

            bucket_n[i - n_max_bucket]['bit_sum'] += bucket_n[i - n_max_bucket + 1]['bit_sum']

            bucket_n[i - n_max_bucket]['timestamp'] = bucket_n[i - n_max_bucket + 1]['timestamp']

            del bucket_n[i - n_max_bucket + 1]


def Count_bit():
    bit_sum = 0

    flag_half = 1

```

```python
    start_time = time.time()
    with open('01stream.txt', 'r') as f:
     for i in range(time_location):
         temp = f.readline()  # 读取文件的值
         if temp:
             Is_due(i + 1)  # 判断是否有桶到期
             if int(temp.strip('\n'))== 1:
                 bucket = {"timestamp": i + 1, "bit_sum": 1}  # 桶的结构
                 bucket_n.append(bucket)
                 Merge()  # 合并大小相同的桶
    for i in range(len(bucket_n)):
        bit_sum += bucket_n[i]['bit_sum']
    bit_sum -= bucket_n[0]['bit_sum'] / 2
    return bit_sum if len(bucket_n) > 0 else 0, time.time() - start_time
bit_sum, bit_time = Count_bit()
bit_act_sum, bit_act_time = Count_bit_act()
print("当前窗口中 1 的估计个数为：%d,运行时间为:%f" % (bit_sum, bit_time))
print("当前窗口中 1 的精确个数为：%d,运行时间为:%f" % (bit_act_sum, bit_act_time))
print("误差值：", abs((bit_act_sum - bit_sum) / bit_act_sum))
```

结果：


for i in range(time_location):
temp = f.readline() # 读取文件的值
if temp:
Is_due(i + 1) # 判断是否有桶到期
if int(temp.strip('\n'))== 1:
bucket = {"timestamp": i + 1, "bit_sum": 1} # 桶的结构
bucket_n.append(bucket)
Merge() # 合并大小相同的桶
for i in range(len(bucket_n)):
bit_sum += bucket_n[i]['bit_sum']
bit_sum -= bucket_n[0]['bit_sum'] / 2
return bit_sum if len(bucket_n) > 0 else 0, time.time() - start_time
C:\ProgramData\Anaconda3\envs\pythonProject1\python.exe C:/Users/29459/PycharmProjects/pythonProject1/main.py
请输入最大相同桶的数量：4
请输入窗口的大小：100
请输入当前时刻（从1开始）：100
当前窗口中1的估计个数为：50,运行时间为:0.001892
当前窗口中1的精确个数为：54,运行时间为:0.000000
误差值： 0.07407407407407407
Process finished with exit code 0

误差率很低。

2 给定一个数据流 A[1,...,n]， 如何估算所有 $A[i]^2$ 的和，若另有一数据流 B[1,...,n]，如何估算 A[i]B[i]之和。

2.1 伪代码

算法 2 对数据流A[1...N]估算$\sum_{i=1}^{n} A[i]^2$

输入: 数据流$A[1...N]$,m=0,r=0,a=0

输出: $\sum_{i=1}^{n} A[i]^2 = m \times (\mathrm{r}^2 - (r-a)^2)$

(**处理j：**)m=m+1
以$\Pr[\beta==1]=\frac{1}{m}$的概率选择随机位置$\beta$
if β==1 **then**
    a=A[j]
    r=0;
end if
if j==find(A,a) **then**
    r=r+a
end if

算法 3 对数据流A[1...N],B[1...N]估算$\sum_{i=1}^{n} A[i]B[i]$

输入: 数据流$A[1...N]$,$B[1...N]$,m=0,r=0,a=0,b=0

输出: $\sum_{i=1}^{n} A[i]^2 = m \times (\mathrm{r}^2 - (r - \sqrt{ab})^2)$

(**处理j：**)m=m+1
以$\Pr[\beta==1]=\frac{1}{m}$的概率选择随机位置$\beta$
if β==1 **then**
    a=A[j]
    b=B[j]
    r=0;
end if
if j==find(A,a)ORj==find(B,b) **then**
    r=r+$\sqrt{ab}$
end if

2.2 分析

上述 find 函数返回对应下标。
当然本题同样也可和 1 题一样，采用抽样的方法进行。

对于上述两种伪代码，算法和频度矩算法相同，只不过将频度显式的设置出来。其中第一个更新步伐为 a，第二个更新步伐为$\sqrt{ab}$。而频度矩算法中，将 a 更新为 j，此处则更改为数组的值 A[j]。频度矩算法相关证明如下所示，对于本题，显然两者是可以互相转化的问题，只需要将步伐更改即可，其他证明相同（见下截图）。由于该算法用$O(logm)$来存储 m 和 r，用$O(logn)$来存储 a 或 b，所以空间复杂度为$O(logm + logn)$。

记A为算法返回时的a的值，相应的r的值记为R

则A的取值为$1, \ldots, n$，R的取值为$1, \ldots, f_j$

计算两个概率：$P[A = j] = \frac{f_j}{m}$

在一个数据流$< a_1, a_2, \ldots, a_m >$中，最后的$A = a_i, i \in [1, m]$（注意这里的相等可以理解为java中的地址相同，而不是值相同）的概率计算思路为：在a_i对应的迭代中，a以$\frac{1}{i}$的概率被更新为a_i，然后在后面所有的迭代中都没有发生改变，这个过程用概率来表示就是$\frac{1}{i} * (1 - \frac{1}{i+1}) * \ldots * (1 - \frac{1}{m}) = \frac{1}{m}$。所以$P[A = a_i] = \frac{1}{m}$，所以自然而然的$P[A = j] = \frac{f_j}{m}$。

计算的思想相似可以得到$P[R = i \wedge A = j] = \frac{1}{m}$

也就是$P[R = i | A = j] = P[R = i \wedge A = j] / P[A = j] = \frac{1}{f_j}$

$$
\begin{aligned}
E[X] &= \sum_{j=1}^{n} E[m(R^k - (R-1)^k)] * P[A = j] \\
&= \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} m(i^k - (i-1)^k) \frac{1}{f_j} \\
&= F_k
\end{aligned}
$$

引理

$kF_1 F_{2k-1} \leq kn^{1-\frac{1}{k}} F_k^2$

求解

$$
\begin{aligned}
Var[X] &\leq E[X^2] \\
&= \sum_{j=1}^{n} \sum_{i=1}^{f_j} m(i^k - (i-1)^k)^2 \\
&= \sum_{j=1}^{n} \sum_{i=1}^{f_j} m(i^k - (i-1)^k)(i^k - (i-1)^k) \\
&= \sum_{j=1}^{n} \sum_{i=1}^{f_j} m(i^k - (i-1)^k) k i^{k-1} \\
&\leq \sum_{j=1}^{n} \sum_{i=1}^{f_j} m(i^k - (i-1)^k) k f_j^{k-1} \\
&= mkF_{2k-1} \\
&= kF_1 F_{2k-1} \\
&\leq kn^{1-\frac{1}{k}} F_k^2
\end{aligned}
$$

算法分析

计算$E[X]=F_k$，计算见附录1

计算$Var[X] \le kn^{1-\frac{1}{k}}F_k^2$，计算见附录2

利用切比雪夫不等式对结果进行检验$P[|X-E[x]|>\epsilon E[X]]<\frac{kn^{1-\frac{1}{k}}}{\epsilon^2}$

评价

算法的方差太大，需要进一步的改进，但是其相应的存储代价为$O(logm+logn)$。

3 设计有效的外存算法，求N个元素中是否存在两个数的和等于K，假定N很大，无法全部放到内存。

3.1 伪代码：

算法 4 求N个元素中是否存在两个数的和等于K

输入: 大数据$A[1...N]$(开始时存在外存)，目标值k，内存容纳数据量M，磁盘块数据量B

输出:

执行归并排序（归并排序过程同课上相同，详细见附I）；
$result$ =0;
for i=1 **to** N/B **do**
    读入M/2B个磁盘块的数据，对应A[i...i+M/2]（这里仅展现大致意思）
    读入M/2B个磁盘块的数据，对应A[N-i*M/2，N-(i-1)*M/2]
    while $k<l$ **do**;
        while $A[k]+A[l]<k$ AND $k<l$ **do**
            k++;
        end while
        while $A[k]+A[l]>k$ AND $k<l$ **do**
            l−−;
        end while
        if $A[k]+A[l]==k$ **then**;
            return 1;
        end if;
    end while
end for;
return 0;

3.2 分析

（1）**解题思路：首先**用外存归并算法进行排序，再用双指针查找是否有等于 K 的两个数。

首先将 N 个元素划分到 M/B 中，相当于每一份划分可以放在内存中分别进行排序。而对于所有的划分，分开的每个队列都能在内存中分一块，然后就可以在内存中进行归并。

双指针包括以下过程：

数组从小到大排序，左指针 l=0，右指针 r=n-1

if(a[l]+a[r]==k) 说明找到了，输出答案，l++，r–

if(a[l]+a[r]<k) 说明找小了 右指针不能变大，所以左指针右移 1 位，l++

if(a[l]+a[r]>k) 说明找大了 左指针不能变小，所以右指针左移 1 位，r–

（2）复杂度分析

本题复杂度分析包括归并排序和双指针找目标数：

① 归并排序的分析如下所示。

buff 中，如果 buff 满，则将其放到外存。

算法 4-1　外存归并排序算法 Mergesort(A，N)

输入： 数组 A，A 包含的块数 N

$n \leftarrow \frac{M}{B}-1$

for $i \leftarrow 0$ to n **do**

if $\frac{N}{n} \leqslant \frac{M}{B}$ **then** //如果 A 的分组块数能够完全放入内存

Sort(A[i]) //对 A 的第 i 个分组直接进行内存排序

else

Mergesort$\left(\mathrm{A}[i], \frac{N}{n}\right)$

Merge(A，N)

Merge(A，N)//N 为数组 A 包含的块数

{

$n \leftarrow \frac{M}{B}-1$//将 A 分为 n 组

length$\leftarrow \frac{N}{n}$//A 的每个分组包含的块数

for $i=0$ to n **do**

Load($A_{i,0}$)；//将 A 每个分组的第 0 个块载入内存

count$\leftarrow$0；//count 是已排序并且输出到外存中的块数

while count$\leqslant N$//每次找最小和第二小的数据项放入缓冲区并输出外存

{

$k_1 \leftarrow$INF //最小元素

$p_1 \leftarrow 0$ //最小元素对应分组下标

$k_2 \leftarrow$INF //次小元素

$p_1 \leftarrow 0$ //次小元素对应分组下标

buff$\leftarrow\varnothing$

//找最小元素，输出外存，并且在对应分组装载新元素到内存

for $i \leftarrow 0$ **to** n

do if $A_{i,\mathrm{pos}[i]} < k_1$

then $p_1 \leftarrow i$

$k_1 \leftarrow A_{i,\mathrm{pos}[i]}$

$A_{p1,\mathrm{pos}[p_1]} \leftarrow$INF

pos[p_1]$\leftarrow$pos[p_1]+1 //更新分组下标

append(buff，k_1)　　　//将 k_1 放到 buff 中

Load($A_{p1,\mathrm{pos}[p_1]}$) //在对应分组装载新的元素到内存

//找次小元素，输出外存，并且在对应分组装载新元素到内存

for $i \leftarrow 0$ to n

do if $A_{i,\mathrm{pos}[i]} < k_2$

then $p_2 \leftarrow i$

$k_2 \leftarrow A_{i,\mathrm{pos}[i]}$

$A_{p_2,\mathrm{pos}[p_2]} \leftarrow$INF

Load($A_{p_2,\mathrm{pos}[p2]}$)

pos[p_2]$\leftarrow$pos[p_2]+1

append(buff，k_2)

Load($A_{p_1,\mathrm{pos}[p_1]}$)

count$\leftarrow$count+1//增加计数器

}

}

2. 外存归并排序的 I/O 复杂度分析

设问题实例数据项个数为 N，每个磁盘块中数据项个数为 B，内存能容纳的数据项个数为 M，首先考虑两个基本参数。扫描一遍数组需要 N/B 次磁盘 I/O。把内存装满需要 M/B 次磁盘 I/O，M/B 也是内存里能装下的磁盘块数。

下面对外存归并排序的 I/O 复杂度进行分析。

定理 4-1 *外存归并排序的磁盘 I/O 复杂度是* $O(N/B\times\log_{M/B}N/B)$。

证明 归并排序第一阶段先把第一组的 M/B 个磁盘块调入内存并排序，然后再把第二组的 M/B 个磁盘块调入内存并排序，直到所有分组调入内存排序完毕。每一组需要 M/B 次 I/O，共 N/M 组，在这一个阶段需要 $M/B\times N/M=N/B$ 次磁盘 I/O。

第二阶段相当于把每组的第一磁盘块放入内存，每一块都要放入内存一次，需要 N/B 次磁盘 I/O。同样，第三阶段还是需要 N/B 次磁盘 I/O，接下来各轮也都是同样的过程，都是 N/B 次 I/O。故每一轮的复杂度都是 $O(N/B)$。

每一轮需要的磁盘 I/O 次数已知，下面计算排序所需的轮数，从而计算出算法的 I/O 复杂度。

第一轮中每次排好序的数组中磁盘 I/O 是 M/B，因为每个磁盘块放在内存中一次。那么第二轮每一次要归并 $M/B-1$ 数组，因为内存要留一个作为缓冲区，则排好的数组大小是 $(M/B-1)\times M/B$。在第三轮中，又要归并第二轮中的排好序的数组，则排好的数组大小是 $(M/B-1)\times(M/B-1)\times M/B=(M/B-1)^2\times M/B$。

依此类推，假设归并第 k 轮结束，则在第 k 轮中排好序的数组大小是 $(M/B-1)^{k-1}\times M/B$。又因为整个数据块的大小是 N/B，所以在第 k 轮中排好序的数组大小 $(M/B-1)^{k-1}\times M/B=N/B$。

整理得到

$$(M/B-1)^{k-1}\times M/B=N/B$$

$$(M/B-1)^{k-1}=N/M$$

$$k=\log_{M/B-1}N/M=O(N/B\times\log_{M/B}N/B)$$

这样每轮的 I/O 数是 $O(N/B)$，所需的轮数为 $O(N/B\times\log_{M/B}N/B)$，则归并排序的 I/O 复杂度为 $O(N/B)\times O(N/B\times\log_{M/B}N/B)$，其中 log 底数中的 M 可以换成 B，$\log N/M$ 可以看成 $\log N/B\times B/M$，而 $\log B/M=1$，是一个低阶项，在复杂度方面可以不考虑，所以最后的结果是 $O(N/B\times N/B\times\log_{M/B}N/B)$。 ■

② 双指针找目标数的复杂度显然是一次完整的扫描，其复杂度为 O（N/B）故总的复杂度和归并排序相同。

为

$O(N/B\times N/B\times\log_{M/B}N/B)$。

附录

1）算法截图

算法 1 计算滑动窗口中1的个数

输入: 目标长度k,数据流$A[1...N]$

输出: 滑动窗口中1的近似个数result

$B[1...k] = 0$;

$result$ =0;

y =0;

for i=1 **to** k **do**

    B[i]=A[i];

end for;

for i=k+1 **to** N **do**

    A[i]以$\frac{k}{i}$的概率随机替换B[1..k]中的一个元素

end for;

for i=1 **to** k **do**

    if $B[i]$==1 **then**;

        y ++;

    end if;

end for;

$result = \frac{y \times N}{k}$

return $result$

算法 2 对数据流A[1...N]估算$\sum_{i=1}^{n} A[i]^2$

输入： 数据流$A[1...N]$,m=0,r=0,a=0

输出： $\sum_{i=1}^{n} A[i]^2 = m\times(\mathrm{r}^2-(r-a)^2)$

(**处理j：**)m=m+1
以Pr[β==1]=$\frac{1}{m}$的概率选择随机位置β
if β==1 **then**
    a=A[j]
    r=0;
end if
if j==find(A,a) **then**
    r=r+a
end if

算法 3 对数据流A[1...N],B[1...N]估算$\sum_{i=1}^{n} A[i]B[i]$

输入： 数据流$A[1...N]$,$B[1...N]$,m=0,r=0,a=0,b=0

输出： $\sum_{i=1}^{n} A[i]^2 = m\times(\mathrm{r}^2-(r-\sqrt{ab})^2)$

(**处理j：**)m=m+1
以Pr[β==1]=$\frac{1}{m}$的概率选择随机位置β
if β==1 **then**
    a=A[j]
    b=B[j]
    r=0;
end if
if j==find(A,a)ORj==find(B,b) **then**
    r=r+$\sqrt{ab}$
end if

↵

算法 4 求N个元素中是否存在两个数的和等于K

输入： 大数据$A[1...N]$(开始时存在外存)，目标值k，内存容纳数据量M，磁盘块数据量B

输出：

执行归并排序（归并排序过程同课上相同，详细见附I）;
$result$ =0;
for i=1 **to** N/B **do**
    读入M/2B个磁盘块的数据，对应A[i...i+M/2]（这里仅展现大致意思）
    读入M/2B个磁盘块的数据，对应A[N-i*M/2，N-(i-1)*M/2]
    while $k < l$ **do**;
        while $A[k]+A[l] < k$ AND $k < l$ **do**
            k++;
        end while
        while $A[k]+A[l] > k$ AND $k < l$ **do**
            l−−;
        end while
        if $A[k]+A[l] == k$ **then**;
            return 1;
        end if;
    end while
end for;
return 0;

2）参考资料

[1]王宏志. 大数据算法[M]. 机械工业出版社, 2015.
[2]https://blog.csdn.net/qq_41445357/article/details/91340828
[3]https://blog.csdn.net/tongtao_123/article/details/79959141
[4]https://www.zhihu.com/question/37511567/answer/154056206
[5]https://blog.csdn.net/weixin_30827435/article/details/116184218
[6] https://www.cnpython.com/pypi/dgim
[7] http://infolab.stanford.edu/~ullman/mmds/ch4.pdf
[8] https://blog.csdn.net/dm_ustc/article/details/46011557
[9]https://blog.csdn.net/suibianshen2012/article/details/51923477?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-4.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-4.control