Expriment 1

PB17111623 范睿

2019 年 11 月 17 日

1 数据库查询v1

1.1 思路

INSERT: 一次性读入所有INSERT命令并解码所有被插入的数字
FIND: 读入第一个FIND时，对所有数据进行快速排序
利用二分查找返回FIND的结果
EXIT: 遇到EXIT退出

1.2 算法

数据库查询v1 1 Database
输入: *Instructions*全部指令
输出: *OUT*查找结果集合

```
while InsType is not EXIT do
    if InsType is INSERT then
        //Insert the id and the attribute without sorting
    else if InsType is FIND then
        if firstime_finding then
            Random_Sort(database)
        end if
        if !Binary_Search(database, queried_data, 0, database.length) then
            PRINT(-1)
        end if
    end if
end while
```

1.3 复杂度分析

二分查找复杂度为$\mathcal{O}(\log n)$
快速排序平均时间复杂度为$\mathcal{O}(n \log n)$，最坏时间复杂度为$\mathcal{O}(n^2)$
此算法平均时间复杂度为$\mathcal{O}(n \log n + \log n) = \mathcal{O}(n \log n)$
最坏时间复杂度为$\mathcal{O}(n^2 + \log n) = \mathcal{O}(n^2)$

2 股票

2.1 思路

此题本质上是在求一个数串逆序数的大小
利用分治法的算法特性：
每次在分治法merge时，传入的p、q、r分别是左串的左端点、左串的右端点（q+1为右串的左端点），右串的右端点。其中左串与右串都是从左到右排好序的串。
merge每次选择整个串中最小的元素加入最后的序列中。如果选择的是右串的元素，那么说明左串的未选择的所有元素比右串的被选择的元素都大，那么在最后结果sum中加上左串中所有的未选择的元素个数；如果选择的是左串的元素，不改变sum的值。（因为左串天然地存在于右串的前面）

2.2 算法

股票算法 2 Merge

输入： *Array*数组，*p*左串左端点，*q*左串右端点，*r*右串右端点
输出： *sum* p与r间的逆序数

```
1: sum = 0
2: for i = p to q do
3:     Left[i-p] = Array[i]
4: end for
5: for i = q + 1 to r do
6:     Right[i-q-1] = Array[i]
7: end for
8: i=0
9: j=0
10: for k = p to r do
11:     if Left[i] ≤ Right[j] then
12:         Array[k] = Left[i]
13:         i++
14:     else
15:         Array[k] =Right[j]
16:         j++
17:         sum += Left.length - i - 1
18:     end if
19: end for
```

2.3 复杂度分析

此算法仅在分治法的基础上加了几条命令，所以其复杂度与分治法的复杂度相同。
所以此算法的复杂度为$\mathcal{O}(n \log n)$

3 弹幕游戏

3.1 思路

先将数字串利用快速排序排好序
再遍历计算相邻两个数字的差，同时找出最大的差值。

3.2 算法

弹幕游戏算法 3 Bullet Game

输入: n数列长度，$Array$坐标

输出: d最大值

```
1: Quick_Sort(Array, 0, Array.length)
2: max = 0
3: for i = 1 to Array.length − 1 do
4:     if max ≤ Array[i+1]-Array[i] then
5:         max = Array[i+1] - Array[i]
6:     end if
7: end for
8: d = max
```

3.3 复杂度分析

快速排序平均时间复杂度为$\mathcal{O}(n \log n)$，最坏时间复杂度为$\mathcal{O}(n^2)$
遍历时间复杂度为$\mathcal{O}(n)$
所以此算法平均时间复杂度$\mathcal{O}(n \log n + n) = \mathcal{O}(n \log n)$
最坏时间复杂度为$\mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$

4 银行卡

4.1 思路

此题本质上是为了计算出每一个大小为k的窗口中最大的元素。
利用双端队列解决：
定义一个双端队列：其队3头可以出队，队尾可以入队+出队。
理想情况下，在窗口在第p个位置时，从p开始往后k个元素的最大值始终处于队头。
①第一个任务是初始化此队列，使当窗口在第0个位置（窗口框住0...k+1的元素）时，队列具有上述性质。
实现方案：用i遍历数组0到k-1的元素：
1.若队列为空，将i入队；
2.若队列不为空，且若数组中第i个位置的元素比第队尾个位置的元素大，从队尾出队；
3.若队列不为空，且若数组中第i个位置的元素比第队尾个位置的元素小，将i入队。(表示第i个位置的元素在前面所有元素滑出窗口后有可能成为最大值)
②第二个任务是遍历数组，找出窗口在每个位置时的窗口内最大元素，假设窗口在第p个位置（p=0...n-k+1）：
1.数组中第p+k-1个元素为每次窗口移动后新进来的元素，令每次新进来的元素为x，若x大于数组中队尾位置的元素，从尾部出队。
2.数组中第p+k-1个元素为每次窗口移动后新进来的元素，令每次新进来的元素为x，若x小于数组中队尾位置的元素，将p+k-1入队。
3.若队头元素小于i，说明队头元素已经滑出窗口，从队头出队。
4.若队头元素大于等于i，打印数组中队头位置的值。

4.2 算法

4.3 复杂度分析

第一个任务初始化队列的复杂度为$\mathcal{O}(k)$
第二个任务复杂度为$\mathcal{O}(n)$
因此此算法复杂度为$\mathcal{O}(n+k)$

银行卡算法 4 VISA

输入: n数列长度，k窗口大小，$Array$每天的钱数

输出: d最大值

```
for i = 1 to k do
    if Q is not empty then
        EnQ(Q, i)
    else
        while Q is not empty and Array[Q.back]≤ Array[i] do
            DeQ_rear(Q)
        end while
        EnQ(Q, i)
    end if
end for
for i = 1 to n − k do
    if i is not 1 then
        if Q is not empty then
            EnQ(Q, i)
        else
            while Q is not empty and Array[Q.back]≤ Array[i] do
                DeQ_rear(Q)
            end while
            EnQ(Q, i)
        end if
    end if
    while Q.front < i do
        DeQ_front(Q)
    end while
    Print Array[Q.front]
end for
```