Fraudulent Activity Notifications

Zifan Gu
Wofford College
429 North Church Street
Spartanburg, SC 29303-3663
guz@email.wofford.edu

ABSTRACT

IN THIS PAPER WE DESCRIBE AN ALGORITHM FOR FINDING THE CORRECT NUMBER OF FRAUDULENT ACTIVITY NOTIFICATIONS OVER A PERIOD OF DAYS. THE SOLUTION REQUIRED FINDING THE MEDIAN IN AN ARRAY EFFICIENTLY TO HANDLE LARGE INPUT CASES. THE ALGORITHM'S EFFICIENCY FOR FINDING MEDIAN IS LOGARITHMIC TIME AND THE SOLUTION RUNS IN LOG-LINEAR TIME WITH LINEAR SPACE COMPLEXITY, AMORTIZED.

Keywords

Counting sort; Binary search; logarithmic time find median

1.INTRODUCTION

The C++ standard template library sorting function use a three part hybrid sorting algorithm, possessing a log-linear time complexity [1]. This paper examines a linear time and space sorting algorithm and finds the median in an unsorted array with logarithmic time, under certain conditions. The algorithm is examined in hypothetical banking accounts to monitor clients' daily spending habits and raises flags when suspicious activities occur.

2.PROBLEM STATEMENT

A local bank has a simple policy for warning clients about the potential fraudulent activities in their accounts: if the amount spent on any given day equals or exceeds twice the client's median spending for a trailing number of days, a fraudulent activity notification will be sent. Given the number of trailing days *d* and the client's total daily expenditure for *n* days, find the number of times the client will receive notifications over *n* days [2].

The number of daily expenditures days *n* will be between 1 and 20,000. The number of trailing days *d* will be between 1 and *n*. Each daily expenditure value expenditure*[i]* will be between 0 and 200. All three input ranges are inclusive.

Example input: n equal 9 days; d equals 4 days; input array is [3, 2, 4, 15, 10, 11, 23, 18, 4]. The algorithm returns 2. Fraudulent warnings occur on day 5 and 7, with median value of the trailing 4 days 3.5 and 10.5, respectively. Details of this example are described in section 5.

3.ANALYSIS

This problem requires us to sort the array of *d* elements, repeated *n - d* times. Our initial approach is to use definite loops to set up the array of *d* elements, then sort the array, and find median value by indexing. This proves to be inefficient as it has quadratic time algorithm complexity.

The brute force approach would require sorting the expenditure in groups of *d*, repeated *n-d* times. The C++ standard template library function *sort(first, last)* runs in $\Theta(n \cdot \log(n))$. Performing *n-k* sorts would have a time complexity of $\Theta((n-k) \cdot n \cdot \log(n))$. For this particular problem, when input sizes are as big as 20,000 elements, the quadratic brute force algorithm is clearly too inefficient to handle the elements.

Because the daily expenditure value has a constraint of between 0 to 200, an integer sorting algorithm, c*ounting sort,* can be implemented with $\Theta(n)$ time complexity [3]. The algorithm counts the number of objects that has distinct key values. The count is then stored in an auxiliary array. Finally, the sorting is accomplished by mapping the count from the index of the auxiliary array [4].

One finding is that after sorting through the array one time, the sequential arrays are almost sorted, with a difference of one element. That is, at any given instance *a,* only *expenditure[a]* needs to be removed from the sorted array and *expenditure[a+d]* needs to be inserted into the array for median processing. There is no need to sort the array *n-d* times.

4.ALGORITHM

This algorithm starts by using c*ounting sort* for the window of *d* elements. Then the algorithm compares the median value with the first element to the right of the processing window. If suspicious criterias match, *fraud_count* increments. The algorithm then calls *replace()* to search the first element in the window in the sorted array and replace that value with the first element to the right of the window at appropriate indexes. This ensures *pro_array* to be sorted at all times, meaning finding a value is logarithmic time using *binary search*. Inserting into *pro_array* is amortized logarithmic time, since finding the corresponding element is logarithmic, but in worst case the insertion operation would be linear if the index is placed at the front of the array.

The brute force approach with quadratic time complexity only works for the HackerRank sample inputs, as the rest of test cases were terminated due to timeout. The modified algorithm utilizing *binary search* and insertion passes all HackerRank tests.

4.1.PSEUDOCODE

This algorithm uses *binary search* to efficiently remove a value in the array and insert a value at the corresponding index to keep the array sorted. The second algorithm *activityNotifications*() relies heavily on *replace*() to traverse to each *expenditure* values to update *pro_array*.

```
ALGORITHM replace(pro_array[0..n-1], a, b)
  // Inputs:   pro_array, a sorted array of n integers
  //           a, an element in pro_array to be deleted
  //           b, an element inserted at a position to keep
  //           pro_array sorted
  // Output:   sorted array with b replacing a
  x = binary_search(pro_array, a)
  // removes the element at index x in the pro_array
  pro_array.remove(x)
  // return the index that is the first element which has a
  // value no less than b
  y = lower_bound(pro_array, b)
  // insert the element b at index y
  pro_array.insert(y, b)
  return pro_array
```

$$\sum_{i=0}^{n-d} log_2(d) + \sum_{i=0}^{d} 1 + \sum_{j=0}^{d} 1 \approx (n - d)log_2(d) + d + d \ = (n - d)log_2(d) + 2d$$

Figure 1. Time complexity of activityNotifications. [7]

```
ALGORITHM activityNotifications(expenditure[0..n-1], d)
  // Inputs:   expenditure, an array of n integers
  //           d, the number of trailing
  // Output:   Total number of times a notification was
  //           sent over n days
  fraud_count ← 0
  pro_array [0…d]
  for i ← 0…d:
    pro_array[i] ← expenditure[i]
  CountingSort(pro_array)
  for j ← 0…n - d:
    if expenditure[d+j] ≥ 2 • median(pro_array):
      fraud_count ← fraud_count + 1
    replace(pro_array, expenditure[j], expenditure[j + d])
  return fraud_count
```

4.2.TIME AND SPACE COMPLEXITY

There are three operations in this algorithm accounting for the time complexity. The first is initializing *pro_array,* time complexity $\Theta(d)$. Counting sort has time complexity $\Theta(d)$, as mentioned in section 3. The last *for* loop traverse through the array $n - d$ times, each time with an operation *replace()* running with $\Theta(\log(d))$ time complexity. Detailed derivation of the algorithm complexity is derived in Figure 1.

The space complexity is influenced by the counting sort algorithm. It finds the maximum value in an array and allocates that amount of space. Since the input array values are contained to be between 0 and 200, the space complexity is $\Theta(200)$. That can be simplified to $\Theta(1)$, meaning constant space complexity.

5.EXAMPLE

In Section 2, an input sample of [3, 2, 4, 15, 10, 11, 23, 18, 4] was given. In this section we will explain in detail why the output is two [4].

For the first d = 4 days, the customer receives no notifications because the bank has insufficient transaction data. *fraud_count* = 0

On the fifth day, the bank has d = 4 days of prior transaction data [3, 2, 4, 15], and median is (3+4)÷2 equal 3.5. The client spends 10 dollars, which increases the *fraud_count* because 10 < 2 * 3.5. *fraud_count* = 1

expenditure **pro_array**

$\boxed{3\ 2\ 4\ 15}$ 10 11 23 18 4 | 3 | 2 | 4 | 15 |

On the sixth day, transaction data [2, 4, 15, 10] has a median of (4+10)÷2 equals 7. The client spends 10 dollars, which does not increase *fraud_count* because 10 < 2 * 7.

expenditure **pro_array**

3 $\boxed{2\ 4\ 15\ 10}$ 11 23 18 4 | 2 | 4 | 10 | 15 |

On the seventh day, transaction data [4, 15, 10,11] has a median of (10+11)÷2 equals 10.5. The client spends 23 dollars, which increases *fraud_count* because 23 > 2 * 10.5. *fraud_count* = 2

expenditure **pro_array**

3 2 $\boxed{4\ 15\ 10\ 11}$ 23 18 4 | 4 | 10 | 11 | 15 |

The window continues to shift to the right. Each time calculate the new median and compare it with the first element to the right of the window. The program exits the loop after comparing the ninth day spending information. The trailing four days had a median spending of (11+18)÷2 equal 14.5 dollars, less than twice of the ninth day spending of 4 dollars. At the end, the program records day 4 and 6 having the suspicious criteria and returns 2 as the number of potential fraudulent activities.

expenditure **pro_array**

3 2 4 15 $\boxed{10\ 11\ 23\ 18}$ 4 | 10 | 11 | 18 | 23 |

6.IMPLEMENTATION

The implementation of this algorithm relies heavily on the vector class in the standard C++ template library. The biggest advantage of using *vector* over arrays or lists is that vectors can be allocated dynamically and stored automatically by the container [5].

One of the parameters passed in is already *vector*. To load *d* datas into an additional array not only decreases efficiency but also complicates of the algorithm. In addition, inside the main loop functions *lower_bound()*, *erase()*, *emplace()* are exclusive for the *vector* class. *lower_bound()* uses *binary search*, a remarkably efficient algorithm to find values in sorted arrays, with the complexity of logarithmic time and constant space. An example of using *binary search* is shown in Figure 2. To implement the entire algorithm using arrays would require additional function declaration for *binary searching* and array reordering because of the insertions to static arrays, both which would increases memory space and program running time.

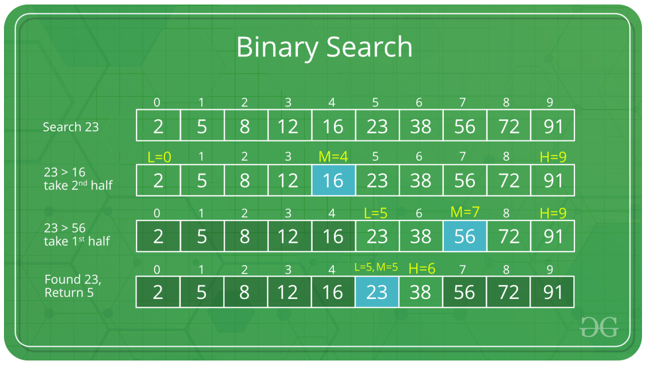

Figure 2. Binary search algorithm to find 23 in an array.[6]

7.CONCLUSIONS

In terms of explaining what a person or client wants the algorithm to accomplish, it makes sense to use small examples for others to easily understand. It would be unrealistic and unnecessary to use examples that has a size of, say 20,000, the upper bound of this challenge. However, in the banking industry, it is common for a bank to have tens of thousands of clients, each with thousands of transactions. The amount of any calculations (e.g. monthly report, cash back matching, interests charged, etc.) required is well over 20,000. At that time, a log-linear time complexity algorithm will make a huge difference than a quadratic time algorithm.

This challenge gave us a valuable lesson and insight. An exponentially increasing amount of industries are incorporating computer science with their departments and product. It is essential for us programmers to note that correctness is only the very basic requirement of an algorithm, not the only requirement. Our algorithms should not be designed to only handle small class sample, but also possessing industrial standards: accuracy, speed, and simplicity.

8.ACKNOWLEDGEMENTS

I would like to thank the senior software engineer and their interns for giving me the idea of using *binary search* to decrease the program run time and algorithm complexity to log-linear. They currently manage databases in the banking industry and asked not to reveal their identity. I would also like to thank Dr. David Sykes for noting that the algorithm only need to sort the input list once, as also to reduce program run time.

9.REFERENCES

1. Wikipedia: 2020. https://en.wikipedia.org/wiki/Sort_(C%2B%2B). Accessed 2020-04-30.
2. HackerRank: 2020. https://www.hackerrank.com/challenges/fraudulent-activity-notifications/problem. Accessed 2020-04-05.
3. GeeksForGeeks: 2020. https://www.geeksforgeeks.org/counting-sort. Accessed 2020-04-07.
4. Programiz: 2020. https://www.programiz.com/dsa/counting-sort. Accessed 2020-4-12.
5. CPlusPlus: 2020. http://www.cplusplus.com/reference/vector/vector/. Accessed 2020-04-13.
6. GeeksForGeeks: 2020. https://www.geeksforgeeks.org/binary-search/ Accessed 2020-04-16.