Design and Analysis of Algorithms

Theory Project

Fractional Cascading

An algorithmic approach

By

IMT2019051 Mani Nandadeep Medicharla

IMT2019063 R Prasannavenkatesh

IMT2019525 Vijay Jaisankar

Contents

1 Abstract

In this paper, we investigate the *Fractional Cascading* technique in building range trees and for fast searching of an element in multiple arrays. In this venture, we introduce and examine *Linear Search*, *Binary Search*, *Bridge Building* and *Auxiliary lists.* We also look at some of the *applications* of this technique, and suggest data structures for its efficient realisation.

2 Problem Statement

Fractional cascading: You are given an input of k ordered lists of numbers, each of size n as well as a query value x. The problem's output is to return, for each list, True if the query value appears in the list and False if it does not. For example, if the input is:

(a) List L_1: $[3, 4, 6]$

(b) List L_2: $[2, 6, 7]$

(c) List L_3: $[2, 4, 9]$

and the query value is 4, then the expected output is [*True*,*False*,*True*].
Give an algorithm to solve the fractional cascading problem.

3 Brute Force

3.1 Linear Search

Linear search is the most basic search technique, wherein we sequentially compare each array element to the target element. In the worst case of the target element not coinciding with *any* list element, the algorithm would reach the end of the list and we would report an unsuccessful search. As each element is compared at most once, the time complexity is $O(n)$, where n is the size of the list.

This algorithm forms the basis of the simplest solution to our problem: We just run linear search on each of the k lists. If we have q queries, this takes $O(q \cdot k \cdot n)$ time, which is a lot and in real world situations, if k, q, and n are even moderately large, the time taken would become astronomical.

```
Data: k arrays of size n and a query element x
Result: Boolean array regarding whether element is present in the
        indiced array or not
Function LinearSearch(Array,x):
    for $i \leftarrow 0$ to $n$ do
        if Array[i] == x then
            return True;
        end
    end
return False
Function Main:
    output = [] ;
    for $i \leftarrow 0$ to $k$ do
        output.append(linearSearch(input[i],x);
    end
return 0
```

Note that, however, this approach does not take into account any relevant information given to us in the question which can speed up this algorithm. It is given that the *lists are sorted*, so we can exploit this property and employ a faster searching technique to solve this problem in a better way: *Binary Search*

4 Improved Brute Force

4.1 Binary Search

Binary search is another searching algorithm that works correctly only on sorted arrays.
It begins by comparing the target element with the element at the middle of the list.

- If they are equal, we have found the target in the list
- If the target is larger, and as the list is sorted, we must now turn our attention to the *right half* of the list

- Similarly, if the target is smaller, we must focus on the *left half* of the list.

In the worst case, Binary Search will take $O(\log n)$ comparisons, where n is the size of the list.

To improve the performance of the Brute Force subroutine, we replace the Linear Search subroutine with the aforementioned Binary Search subroutine. If we have q queries, this takes $O(q \cdot k \cdot \log n)$ time, which is certainly a lot better than the initial brute force algorithm, but can yet be improved further.

```
Data: k arrays of size n and a query element x
Result: Boolean array regarding whether element is present in the
        indiced array or not
Function BinarySearch(Array,x,left,right):
    if $right \geq left$ then
        mid=$\frac{(left+right)}{2}$;
        if Array[mid] == x then
            return True;
        end
        if Array[mid] > x then
            BinarySearch(Array,x,left,mid-1);
        end
        else
            BinarySearch(Array,x,mid+1,right);
        end
    end
    else
        return False;
    end
end Function
Function Main:
    output = [] ;
    for $i \leftarrow 0$ to $k$ do
        output.append(BinarySearch(input[i],x);
    end
return 0
```

5 Bridge Building

5.1 Introduction to bridges

A *bridge* is a pointer from an element a_i of A_i to an element a_j of A_{i+1} where $|a_i - a_j|$ is **small**, where A is the list and a represents an element of the array.
By *small*, we mean an element that is either of the same value, or with the smallest difference to the one considered as reference.

Once we locate the answer to a query in one array, we should be able to **follow a bridge** to a key that is close to the answer in the next array.
In the best case, we follow a bridge from the answer to the query in A_i to a key in A_{i+1} and then from there locate the answer in A_{i+1}, all in constant time. If we can do *this*,then once we have the answer in A_1 we can find the answer in the remaining $k-1$ sorted arrays in $O(k)$ time complexity.

From a technical standpoint, we *implement* this method as follows:
For every element e in the first array, give e a pointer to the element with the *same value* in the second array or if the value doesn't exist, the *predecessor*. This is called bridge building between A_i and A_{i+1}. Then, once we've found the item in the first array, we can just follow these pointers down in order to figure out where the item is in all the other arrays. To find the answer in A_1, we can just use a balanced binary search tree,thus making the overall time complexity of our algorithm $O(\log n + k)$ per query.

5.2 Algorithm

```
Data: k arrays of size n and a query element x
Result: Boolean array regarding whether element is present in the
        indiced array or not
Function BuildBridges(Array,x):
    for $i \leftarrow 0$ to $k - 1$ do
        for $j \leftarrow 0$ to $n$ do
            Build bridge from Array[i][j] to Array[i+1][x] where
             |Array[i+1][y] - Array[i][j]| is small. In this approach, if
             both predecessor and successor exists, then we take
             predecessor first.
        end
    end
end Function
Function Main:
    output = [] ;
    BuildBridges(input,x) ;
    output.append(BinarySearch(Array[i],x)) ;
    Once the element is found in the first array, follow the bridge path
     till the final array and append it to the output. ;
return 0
```

5.3 Shortcomings

This method seems like a very interesting and efficient alternative to solve this problem. However, there are some glaring weaknesses to this approach, the most important one being the fact that certain classes of inputs render this method useless.
In particular, if a later list is completely *in between* two elements of the first list, we have to redo the entire search, as the pointer pre-processing gives us no information that we didn't already know.

Let's consider a simple example to elucidate this statement

Consider the case where $k = 2$. Everything would be better if only we could guarantee that the first list contained the right elements to give you useful information about the second array. We could just merge the arrays, but if we did this in the general case we'd end up with a totally merged array of size kn, which is not so good if k is large.

Even if each time we find an answer in the sorted Array A_i, we follow the bridge pointer from it as well as from the key above it we are still left with the entire contents of catalog A_{i+1} to search. This continues through the entire set of k catalogs.If we search each catalog by doing a linear scan from the point at which the bridge told us to begin,the total search time will be O(nk).Even if we build a balanced search tree over all elements in $Ai+1$ that appear between two consecutive bridge pointers from A_i the query time will still be O(klogn) which is similar to performing k binary searches to get the query element.

6 Fractional Cascading

References

[1] Wikipedia article on Fractional Cascading

[2] Practical Demo Application to demonstrate Fraction cascading

[3] Opengenus reference on Fraction Cascading

[4] An essay on fractional-cascading by arpit bhayani