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 which is used 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 *Fractional Cascading.* We also look at some of the *applications* of this technique, and give a video demonstration of its capabilities.

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.

3.2 Pseudocode

```
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.

4.2 Proof of correctness

We prove the correctness of binary search by the *method of strong induction*

Let $p(k) :=$ Binary search works on an array of size k

1. **Base Case**
   - If $k = 1$, $p(1)$ is trivially true; as either $array[mid] = x$, or not
   - Hence, $p(1)$ is true
2. **Induction Hypothesis**
   - Let $p(1) \wedge p(2) \wedge \cdots \wedge p(z) =$ True
   - This means that binary search works for all arrays of size *at most* z.
3. **Induction Step**
   - Now, we look at an array of size $z + 1$. Here, we have three cases as outlined in the algorithm:
     - If $array[mid] = x$, we are done as we can return prematurely from the loop
     - Else, our new search space will become roughly half(*Note: It will be exactly* $\frac{1}{2}$ *of the original size, if the size of the array is a power of 2*). In any case, the size of the "new" array we should search in, is $\leq z$.

- Now, from our *Strong Induction Hypothesis*, we are done as all of their previous cases are true.
- So, $p(1) \wedge p(2) \wedge \cdots \wedge p(z) \rightarrow p(z+1)$

4. **So, we have proved the correctness of the binary search algorithm by strong induction.**

4.3 Pseudocode

```
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 ≥ 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.[1]

Once we locate the position to a query in a array, we should be able to **follow a bridge** to a element 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 the endpoint of the bridge 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* (*Note*: predecessor(x) $= v \in$ Search space where $x - v$ is minimum, and $x > v$.). 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 might be located 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.

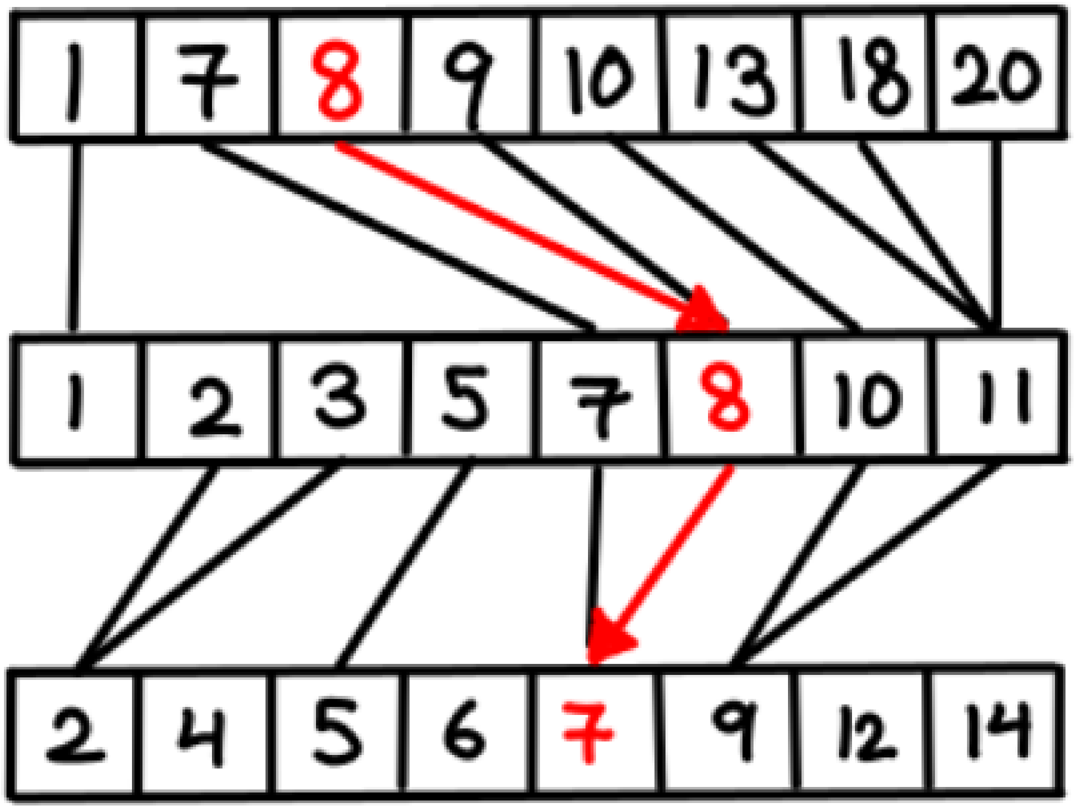

Figure 1: Process of bridge building for a given question.[4]

From the image, we can see the bridge building in action, where the lines between the arrays represent bridges. We can clearly see the predecessor linkages and how we can follow the pointers down and generate the output for all of the arrays.

In this example, we are searching for 8 in all 3 arrays. We can clearly see the path of pointers we should traverse, as outlined in red.Note that, in the last array, the most plausible candidate element is 7 and not 8, so we would return *False* for this array, and hence our overall output will be [*True*,*True*,*False*] as the element 8 is only present in the first two arrays and not in the third one.

5.2 Pseudocode

```
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.

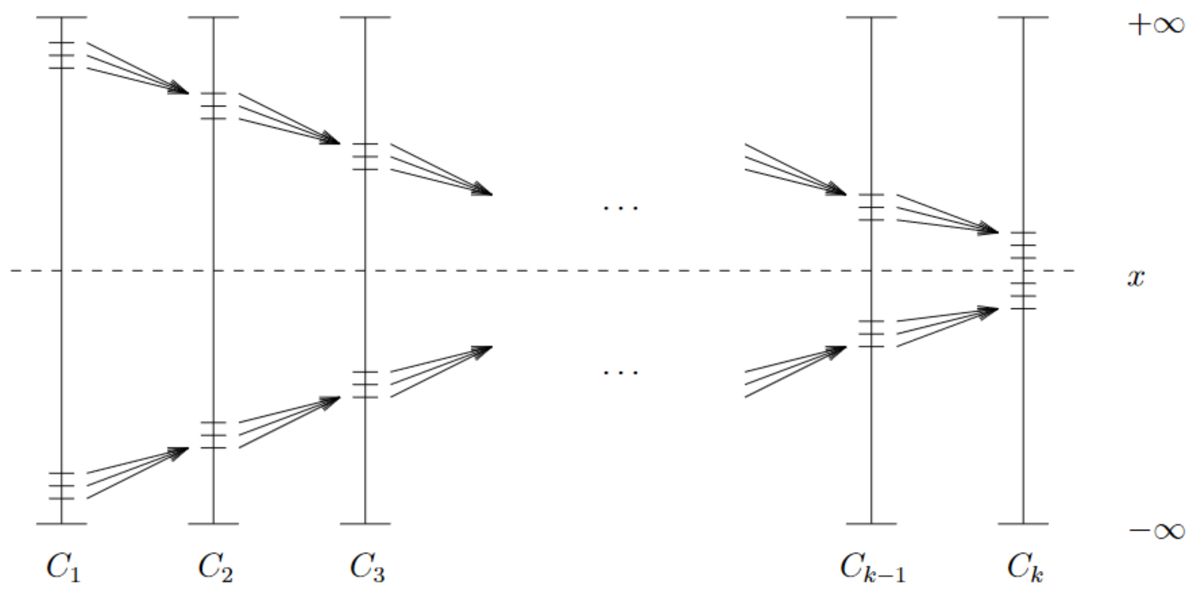

Figure 2: An example where a later list is completely *in between* two elements of the first list illustrated via bridge building.[1]

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. The bridges would be built with only the maximum and minimum of the lower array as endpoints leading us to either search the lower array again or we could just *merge the arrays* naively, but if we did this, we'd end up with an array of size $k \cdot n$, which is not optimal at all, if k is even moderately 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 array A_{i+1} to search. This continues through the entire set of k arrays. If we search each array by doing a linear scan from the point at which the bridge told us to begin,the total search time will be $O(n \cdot k)$ .Even if we build a balanced search tree over all elements in A_{i+1} that appear between two consecutive bridge pointers from A_i the query time will still be $O(k \cdot \log n)$ which is similar to performing k binary searches to get the query element.

6 Fractional Cascading

6.1 Intuition behind Fractional Cascading

Because of the shortcomings of the *Bridge Building* algorithm, i.e the case where every element of the "below" array is in between all the elements of the "above" array.(**Note**: We say that array 1 is "above" array 2 and array 2 is "below" array 1. Therefore, an Array j is below Array i if $j > i$). In this case, we either have to do k binary searches, or we have to merge all n elements of the below arrays recursively and maintain bridges, to get the output. This will warrant a sub-optimal, extra time complexity of $O(k \cdot n)$, as we are going through *each* array element iteratively to merge it.

To avoid this problem of merging all elements and getting $O(k \cdot n)$ time complexity, we start with the *lowest* list in the sequence and select every ith element, where i = $\frac{1}{\alpha}$ and insert it into the array above it while still maintaining sorted order. We then mark that element as *promoted*[3] and keep a pointer from it, to its original position in the bottom list. This operation of taking every ith element and *promoting* it to the array above it is called *cascading*, and since we are only promoting a fraction of the elements, the algorithm is called **fractional cascading**.

For selecting, α, we can choose between a variety of fractions, however $\alpha = \frac{1}{2}$ seems most appropriate because, we will have to compare only 2 elements while searching, after prepocessing. This will also ensure that our time complexity stays low, and easily computable.

6.2 Algorithmic Approach to Fractional Cascading

Let the input be specified by k n-element arrays, $A_1, A_2, \ldots, A_k$, Let the query element value be x. Let $M_1, M_2, \ldots, M_k$ be the new *merged arrays* such that $M_k = A_k$ and $\forall i < k$, M_i is defined as the result of merging M_i with every $\frac{1}{\alpha}$th element of M_{i+1}.

As we're taking $\alpha = \frac{1}{2}$, M_i will be the result of merging M_i with every alternate element of M_{i+1}. For every **cascaded element** of M_i $\forall$ i < k, we keep two pointers from each element which are derived as follows:

- If the element is non cascaded,i.e, If the element is from the Array A_i, then the first pointer points to the smallest cascaded element greater than the element in A_i and the second pointer points to the largest element lesser than the element in A_i.
- If the element has been cascaded, we keep a pointer to the non cascaded predecessor and non

cascaded successor of the element in M_i to be able to efficiently find the next and previous element which is a member of the current array; and also a keep a bridge between M_i and M_{i+1} at the position where it is present in both of the arrays.

Additionally, we add bridges between the pseudo-elements ,i.e, -∞ and ∞ in consecutive arrays.If there is no key of the appropriate type above or below the key,the pointer points to the pseudo-keys at $\pm\infty$, whichever is appropriate.[1] These pointers helps to find the position of the query element x in A_i and also in the cascaded arrays below in $O(1)$ time.

Note: Since we are merging every alternate element of the below list to the current list, we have $|M_i| = |A_i| + \frac{1}{2}|M_{i+1}|$, which in turn ensures that $|A_i| \leq 2n = O(n)$.

After we perform the aforementioned pre-processing, querying x in all k lists is done as follows: First, we make a query for x in M_1 using a binary search in $O(\log n)$. Once we have found the position of x in M_1, we use the *cascaded pointers* to find the position of x in M_2. Generalising this step, once we found the position of x in M_i where i < k, we use the cascaded pointers to find the position of x in M_{i+1}.

To find the location in M_{i+1}, we find the *two neighbouring elements* in M_i that came from M_{i+1} using the pointers we had assigned during the pre-processing phase. Now, these elements will have *exactly one element* between them in M_{i+1}. **Therefore, to find the exact location in M_{i+1}, we just have to do a simple comparison with only the intermediate element**. This is the significance of taking $\alpha = \frac{1}{2}$ as we just have to perform only one comparison, which takes $O(1)$ time, and hence we can retrieve the location of x in A_i from its location in M_i again in $O(1)$ time.

Hence, the time to perform the pre-processing for fractional cascading is O(nk), the total search time per query is $O(k+logn)$ and, the total time taken by the Fractional Cascading algorithm is $O(q(k + logn))$ for q queries, which is an improvement over the previous algorithms.

6.3 Pseudocode

```
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 Fractional_Cascading:
    output = [] ;
    MergedArrays = [] ;
    MergedArrays = MergedArrays.insert(0,all elements of the last
      array) ;
    MergedArrays = MergedArrays.insert(0,merge the below array
      with the above array by only taking alternate elements of the
      below array) ;
    Generate the boundary case predecessors and successors; $-\infty$ and
      $+\infty$ ;
    For every element in the merged arrays, assign locations based on
      the presence of that particular element in $A_i$ and $M_{i+1}$.If the
      element came from the same array, i.e, $A_i$, we keep a pointer to
      the nearest neighbouring element on either side from $M_{i+1}$. If the
      element has been cascaded, we keep a pointer to the non
      cascaded predecessor and successor of the element in $M_i$ and also
      a bridge between $M_i$ and $M_{i+1}$ at the position where it is present
      in both of the arrays.;
    We then check for the position of the target element in the merged
      array, then we follow the pointers down to get the positions of
      the predecessors of the said target element in all of the k arrays.
      Let's call this array $positions[k]$ ;
    The last step is to scan through $positions[k]$, and see if the
      respective predecessor is actually the given target, or not. This
      generates the $[True, False]$ format given in the question and
      append it to the output array.
return output
```

6.4 Example

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

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

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

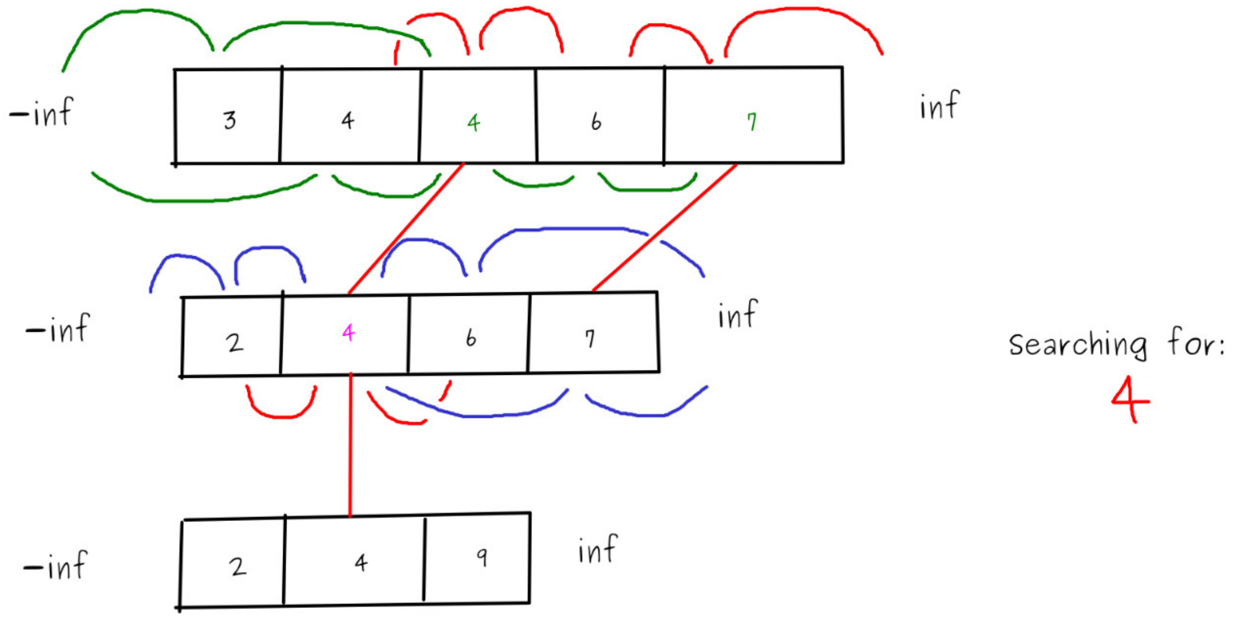

Figure 3: Fractional cascading pre processing for the given question.

The figure gives the final MergedArrays and we can also see how the elements are cascaded. Every cascaded element has a pointer to it's predecessor and successor which was initally present in the same array A_i as shown by the red curved pointers and also has a bridge between merged arrays M_i and M_{i+1} depending on the location of that particular element in both the arrays. The red lines denote the bridges. Also the elements that are present in both A_i and M_i has a pointer to the nearest cascaded elements as shown by the blue and green lines in the figure.

Here, the search query element, $x = 4$. So we start at M_1 and search for 4 using a binary search which will in turn return the first 4 in the array, i.e, the element present in index 1. Now we will follow this index's pointer to the nearest cascaded element which is 4 present at index 2. However, this is a cascaded element from the below arrays and hence will have a bridge to the below arrays. So the algorithm follows the bridge and goes to index 1 of M_2. However, this elements is also cascaded, so we check whether the previous element is equal to 4 or not. The previous element in this case is 2 present at index 0 of M_2 which is not equal to the query element. So we again go to the nearest cascaded element via 2's pointer and reach the next merged array which is M_3. Here the bridge's endpoint is the query element and it will return true and the algorithm is terminated as we have reached the lowest array. Hence the output will be [*True*,*False*,*True*].

We have made a short walkthrough of the above example which can be found on youtube in this Link(click here)[6]

6.5 Proof of Correctness

To prove that our Fractional Cascading algorithm is correct, we perform the following steps.

- We look at another simpler algorithm and prove its correctness, let's call this algorithm X.
- We show that our algorithm and X produce the same output. This forms the proof of correctness of our algorithm.

Note that we already have a candidate for algorithm X; Our *improved brute force algorithm* which uses binary search which we have already proved earlier.

We will follow the concept of universal generalization here; lets consider the r^{th} array of our input sequence consisting of k arrays of size n where r < k. Let's also consider a target element x. Clearly there are two cases

- x ∈ array[r] : let's call this situation 1.
- x ∉ array[r] : let's call this situation 2.

Let's start with situation 1,

Now there are two possibilities here when our control reaches array[r]

- We find element x and it is not cascaded from below then we can say that it returns *true* which is exactly what algorithm X returns in the same situation.
- We find element x but it is cascaded from below,then, our control moves into the nearest non-cascaded neighbours of x. As x ∈ array[r], *native* x will be one of the neighbours of the cascaded x. So, the control gets transferred to the native x, and this case gets *transformed* into the previous case. Hence, we return *true* which is exactly what algorithm X returns in the same situation.

Now, **let's continue with situation 2**. Now, there are two possibilities here when our control reaches array[r]

- We find element x but it is cascaded from below. So we look at its nearest non-cascaded neighbours which are guaranteed not to be x(as $x \notin$ array[r] by supposition). So we can return *false*, which is exactly what algorithm X returns in this situation.
- We find an element which is not equal to x. Now, regardless of whether it's cascaded or not when we traverse to the pointers, we are guaranteed not to find x so we return $false$, which is exactly what algorithm X returns in this situation.

Hence, the output given by the fractional cascading algorithm and the one given by the binary search algorithm is equal. All the cases are exhausted and the algorithm terminates when $M_k = A_k$. This proves that our **algorithm gives correct output**, and hence concludes the proof of correctness.

7 Applications

This technique has various applications[5] in numerous fields.

1. Computational Geometry
   - Half-Range Plane Reporting
   - Explicit Searching
   - Point Location
2. Networks
   - Fast Packet filtering in internet routers.
   - Data distribution and retrieval in sensor networks
3. Linear Range Queries
   - As an accompaniment to *Segment Trees*

References

[1] C.S.252 “Prof.RobertoTamassia” ComputationalGeometry Sem.II, 1992-1993
`http://cs.brown.edu/courses/cs252/misc/resources/lectures/pdf/notes08.pdf`

[2] 6.851:Advanced Data Structures Spring 2012 “Prof.Erik Demaine”
Link to MIT Lecture notes by Prof Erik Demaine on the topic Advanced Data Structures

[3] Fractional Cascading webpage created by Ravi Sinha alias ravix339-zz
`https://ravix339.github.io/FractionalCascading/index.html`
`https://ravix339.github.io/FractionalCascading/Demo.html`

[4] A blog about fractional cascading and bridge building by Edward Z. Yang
`http://blog.ezyang.com/2012/03/you-could-have-invented-fractional-cascading/`

[5] Wikipedia article on fractional cascading
`https://en.wikipedia.org/wiki/Fractional_cascading`

[6] Our youtube walkthrough for fractional cascading
`https://www.youtube.com/watch?v=eUQtziH6cDo`