Searching Algorithms

Content

Fibonacci Search

About

Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

Similarities with Binary Search

1. Works for sorted arrays

2. A Divide and Conquer Algorithm.
3. Has Log n time complexity.

Differences with Binary Search:

1. Fibonacci Search divides given array into unequal parts
2. Binary Search uses a division operator to divide the range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.
3. Fibonacci Search examines relatively closer elements in subsequent steps. So when the input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

Background

Fibonacci Numbers are recursively defined as F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1. First few Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Observations:

Below observation is used for range elimination, and hence for the O(log(n)) complexity.

F(n - 2) approx; (1/3)*F(n) and
F(n - 1) approx; (2/3)*F(n).

Algorithm

Let the searched element be x.

The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of the given array. Let the found Fibonacci number be fib (m'th Fibonacci number). We use (m-2)'th Fibonacci number as the index (If it is a valid index). Let (m-2)'th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for subarray after i, else we recur for subarray before i.

Below is the complete algorithm

Let arr[0..n-1] be the input array and the element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the array has elements to be inspected:
    1. Compare x with the last element of the range covered by fibMm2
    2. **If** x matches, return index
    3. **Else If** x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
    4. **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If a match, return index.

Implementation -

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Searching/Codes/FibonacciSearch.cpp

Time Complexity analysis

The worst-case will occur when we have our target in the larger (2/3) fraction of the array, as we proceed to find it. In other words, we are eliminating the smaller (1/3)

fraction of the array every time. We call once for n, then for(2/3) n, then for (4/9) n, and henceforth.

$$fib(n) = \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right] \sim c*1.62^n$$

$$for\ n \sim c*1.62^n\ we\ make\ O(n')\ comparisons.\ We,\ thus,\ need\ O(\log(n))\ comparisons.$$

Jump Search

About

Like Binary Search, Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

Let's consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). The length of the array is 16. Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.

Steps -

STEP 1: Jump from index 0 to index 4;

STEP 2: Jump from index 4 to index 8;

STEP 3: Jump from index 8 to index 12;

STEP 4: Since the element at index 12 is greater than 55 we will jump back a step to come to index 8.

STEP 5: Perform a linear search from index 8 to get the element 55.

What is the optimal block size to be skipped?

In the worst case, we have to do n/m jumps and if the last checked value is greater than the element to be searched for, we perform m-1 comparisons more for linear search. Therefore the total number of comparisons in the worst case will be ((n/m) + m-1). The value of the function ((n/m) + m-1) will be minimum when m = √n. Therefore, the best step size is m = √**n.**

Implementation -

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Searching/Codes/JumpSearch.cpp

Time Complexity - O(√n)

Auxiliary Space - O(1)

Important points

Works only sorted arrays.

1. The optimal size of a block to be jumped is (√ n). This makes the time complexity of Jump Search O(√ n).
2. The time complexity of Jump Search is between Linear Search ((O(n)) and Binary Search (O (log n)).
3. Binary Search is better than Jump Search, but Jump search has an advantage that we traverse back only once (Binary Search may require up to O(Log n) jumps, consider a situation where the element to be searched is the smallest element or smaller than the smallest). So in a system where the binary search is costly, we use Jump Search.