Fibonacci search algorithm

Source: ***https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms***

1. *What is Fibonacci search algorithm?*

Fibonacci search is an algorithm for searching a sorted array by exploiting the structure of the Fibonacci sequence. It is an example of a divide-and-conquer algorithm that can be used to find an element in a sorted array in a relatively efficient manner.

The algorithm works in the following sequence:

1. First, the algorithm calculates the smallest Fibonacci number that is greater than or equal to the length of the array. Let's call this number F.
2. Next, the algorithm compares the element at the index **`mid = low + F(k-2)`** to the target element. If the element at mid is equal to the target element, the search is successful and the algorithm returns the index of the element.
3. If the element at mid is less than the target element, the algorithm performs a Fibonacci search on the subarray to the right of mid, with the low index set to mid+1 and the high index remaining unchanged.
4. If the element at mid is greater than the target element, the algorithm performs a Fibonacci search on the subarray to the left of mid, with the high index set to mid-1 and the low index remaining unchanged.
5. The algorithm continues this process until the target element is found or it is determined that the element is not present in the array.

Fibonacci search has a time complexity of **`O(log n)`** in the worst case, which is the same as **`binary search`**. However, it has a smaller constant factor and may be slightly faster in practice due to its use of the Fibonacci sequence.

2. *Code example.*

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

// Function to perform Fibonacci search
int fibonacciSearch(std::vector<int>& arr, int x)
{
    // Get the length of the array
    int n = arr.size();

    // Initialize the fibonacci numbers
    int fibM = 0; // (m'th Fibonacci number) - 1
    int fibMm = 1; // (m-1)'th Fibonacci number - 1

    // Fibonacci numbers are 0 and 1 for the first two terms
    int fib = fibM + fibMm;

    // To store the smallest Fibonacci number greater than or equal to n
    while (fib < n)
    {
        fibM = fibMm;
        fibMm = fib;
        fib = fibM + fibMm;
    }

    // Marks the eliminated range from front
    int offset = -1;

    // while there are elements to be inspected
    while (fib > 1)
    {
        // Check if fibM is a valid location
        int i = std::min(offset+fibM, n-1);

        // If x is greater than the value at index i,
        // cut the subarray array from offset to i
```

```
if (arr[i] < x)
        {
            fib = fibMm;
            fibMm = fibM;
            fibM = fib - fibMm;
            offset = i;
        }

        // If x is greater than the value at index i,
        // cut the subarray after i+1
        else if (arr[i] > x)
        {
            fib = fibM;
            fibMm = fibM - fibMm;
            fibM = fib - fibMm;
        }

        // element found
        else {
            return i;
        }
    }

    // comparing the last element with x
    if(fibM && arr[offset+1]==x) return offset+1;

    // element not found
    return -1;
}

int main()
{
    std::vector<int> arr = { 10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100 };
    int x = 85;

    int index = fibonacciSearch(arr, x);
```

```
    if (index != -1)
        std::cout << "Element found at index: " << index << std::endl;
    else
        std::cout << "Element not found" << std::endl;

    return 0;
}
```

```
Output: Element found at index: 8
```

This is because the target element x has a value of 85, and the element with a value of 85 is present at index 8 in the array arr. The function correctly returns the index of the element.

If you change the value of x to a value that is not present in the array, the function will return -1. For example, if you set x to 75, the output will be:

```
Element not found
```

This is because the element with a value of 75 is not present in the array.

3. Code explanation.

The function first calculates the smallest Fibonacci number that is greater than or equal to the length of the array. This is done using a loop that continues until **`fib`** is greater than the length of the array. The value of fib is calculated by adding the previous two Fibonacci numbers together, starting with 0 and 1 as the first two terms.

Next, the function enters a loop that continues until **`fib`** is greater than 1. This loop is the heart of the Fibonacci search algorithm.

Inside the loop, the function calculates the index **`i`** of the middle element in the current subarray being searched. This is done by adding the value of offset to **`fibM,`** and taking the minimum of this value and n-1. The value of offset is initialized to **`-1`**, and is updated each time the search is narrowed to a new subarray.

If the element at index **`i`** is less than **`x`**, the search is narrowed to the subarray to the right of **`i`**, and the value of offset is updated to **`i`**. The values of fib, **`fibM`**, and **`fibMm`** are also updated to reflect the new subarray being searched.

If the element at index i is greater than x, the search is narrowed to the subarray to the left of **`i`**, and the values of fib, **`fibM`**, and **`fibMm`** are updated to reflect the new subarray being searched.

If the element at index **`i`** is equal to **`x`**, the search is successful and the function returns the index of the element.

If the loop ends and fib is equal to **`1`**, the function checks if the last element in the subarray being searched is equal to x. If it is, the function returns the index of the element. If it is not, the function returns **-1** to indicate that the element was not found.

4. Time complexity.

The time complexity of the Fibonacci search algorithm is **`O(log n)`**.

In each iteration of the loop, the size of the subarray being searched is reduced by approximately the ratio of the two previous Fibonacci numbers. Since the ratio of consecutive Fibonacci numbers approaches the golden ratio (1.618...), the subarray size is reduced by approximately 1.618 in each iteration. Therefore, the number of iterations required to search a subarray of size n is approximately log1.618 n = log n / log 1.618, which is O(log n).

Since the Fibonacci search performs a single comparison in each iteration, the time complexity of the algorithm is O(log n). This is an improvement over the time complexity of the binary search algorithm, which is O(log n) as well but requires more comparisons in each iteration.

I wish you happy learning,

Your Rosen Rusev