Academic Year: 2022-2023

Name – Prerna Sunil Jadhav **SAP ID - 60004220127**

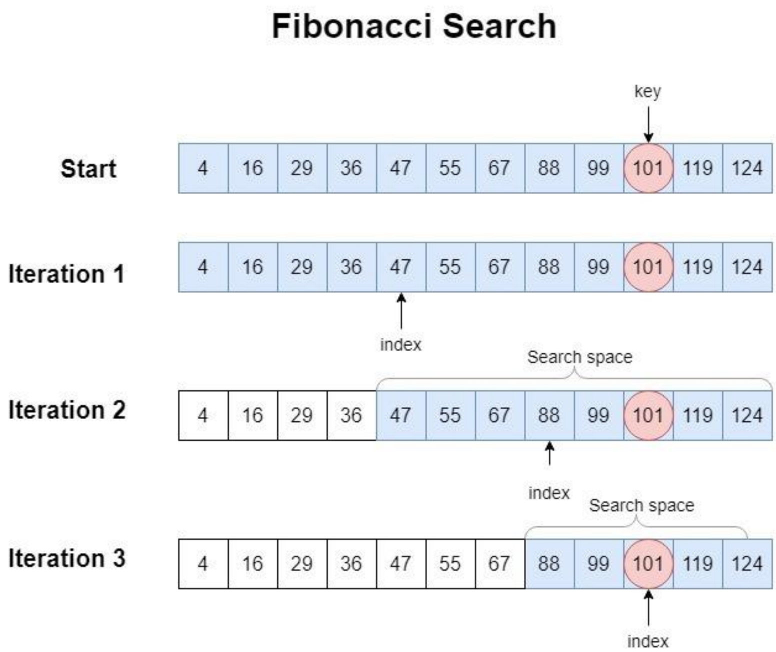
Experiment No – 06

AIM: Implementation of Fibonacci and binary search.

Fibonacci Search

Theory:

- Fibonacci search technique is a method of searching a sorted array using a divide and conquer algorithm that narrows down possible locations with the aid of Fibonacci numbers.
- Compared to binary search where the sorted array is divided into two equal-sized parts, one of which is examined further, Fibonacci search divides the array into two parts that have sizes that are consecutive Fibonacci numbers.



- On average, this leads to about 4% more comparisons to be executed, but it has the advantage that one only needs addition and subtraction to calculate the indices of the accessed array elements, while classical binary search needs bit-shift , division or multiplication, operations that were less common at the time Fibonacci search was first published.
- Fibonacci search has an average- and worst-case complexity of O(log 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 match, return index.
```

Example:

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-

fibMm2	*fibMm1*	*fibM*	*offset*	i=min(offset+fibL n)	*arr[i]*	*Consequence*
5	8	13	0	5	45	Move one down, reset offset
3	5	8	5	8	82	Move one down, reset offset
2	3	5	8	10	90	Move two down
1	1	2	8	9	85	Return i

Program:

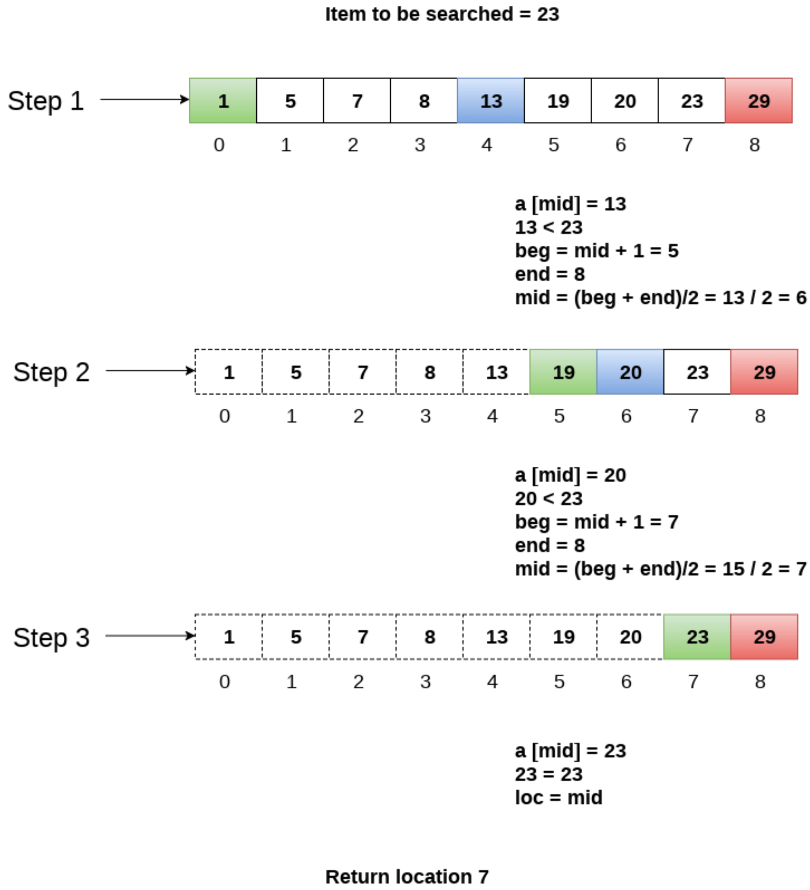
```c
// C program for Fibonacci Search
#include <stdio.h>

// Utility function to find minimum of two elements
int min(int x, int y) { return (x <= y) ? x : y; }

/* Returns index of x if present, else returns -1 */
int fibMonaccianSearch(int arr[], int x, int n)
```

```
{
    /* Initialize fibonacci numbers */
    int fibMMm2 = 0;              // (m-2)'th Fibonacci No.
    int fibMMm1 = 1;              // (m-1)'th Fibonacci No.
    int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

    /* fibM is going to store the smallest Fibonacci
    Number greater than or equal to n */
    while (fibM < n)
    {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

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

    /* while there are elements to be inspected. Note that
    we compare arr[fibMm2] with x. When fibM becomes 1,
    fibMm2 becomes 0 */
    while (fibM > 1)
    {
        // Check if fibMm2 is a valid location
        int i = min(offset + fibMMm2, n - 1);

        /* If x is greater than the value at index fibMm2,
        cut the subarray array from offset to i */
        if (arr[i] < x)
        {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        }

        /* If x is greater than the value at index fibMm2,
        cut the subarray after i+1 */
        else if (arr[i] > x)
        {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }

        /* element found. return index */
```

```
        else
            return i;
    }

    /* comparing the last element with x */
    if (fibMMm1 && arr[offset + 1] == x)
        return offset + 1;

    /*element not found. return -1 */
    return -1;
}

/* driver function */
int main(void)
{
    printf("Prerna Sunil Jadhav - 60004221027\n");
    int arr[] = {10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100, 235};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 45;
    int ind = fibMonaccianSearch(arr, x, n);
    if (ind >= 0)
        printf("\n%d Found at index: %d", x, ind);
    else
        printf("\n%d isn't present in the array", x);
    x = 900;
    int ind2 = fibMonaccianSearch(arr, x, n);
    if (ind2 >= 0)
        printf("\n%d Found at index: %d", x, ind2);
    else
        printf("\n%d isn't present in the array", x);
    return 0;
}
```

OUTPUT:

```
Prerna Sunil Jadhav - 60004221027

45 Found at index: 4
900 isn't present in the array
```



Binary Search

Theory:

- Binary Search is an algorithm that can be used to search an element in a sorted data set.
- By sorted, we mean that the elements will either be in a natural increasing or decreasing order.
- Binary Search works on the principle of Divide and Conquer.
- It divides the array into two halves and tries to find the element K in one or the other half, not both.
- It keeps on doing this until we either find the element or the array is exhausted.
- We start the search at the middle of the array, and divide the array into binary, or two parts.
- If the middle element is less than K, we ignore the left half and apply the same technique on the right half of the array until we either find K or the array cannot be split any further.
- Similarly, if the middle element is greater than K, we ignore the right half and apply the same technique on the left half of the array until we either find K or the array cannot be split any further.

Algorithm:

```
Initialise n = size of array, low = 0, high = n-1. We will use low and high to determine
the left and right ends of the array in which we will be searching at any given time

if low > high, it means we cannot split the array any further and we could not find K. We
return -1 to signify that the element K was not found

else low <= high, which means we will split the array between low and high into two halves
as follows:

o   Initialise mid = (low + high) / 2, in this way we split the array into two halves with
arr[mid] as its middle element
o   if arr[mid] < K, it means the middle element is less than K. Thus, all the elements on
the left side of the mid will also be less than K. Hence, we repeat step 2 for the right
side of mid. We do this by setting the value of low = mid+1, which means we are ignoring
all the elements from low to mid and shifting the left end of the array to mid+1
```

Academic Year: 2022-2023

```
o   if arr[mid] > K, it means the middle element is greater than K. Thus, all the elements on the right side of the mid will also be greater than K. Hence, we repeat step 2 for the left side of mid. We do this by setting the value of high = mid-1, which means we are ignoring all the elements from mid to high and shifting the right end of the array to mid-1
o   else arr[mid] == K, which means the middle element is equal to K and we have found the element K. Hence, we do not need to search anymore. We return mid directly from here signifying that mid is the index at which K was found in the array
```

Program:

```c
// C program to implement recursive Binary Search
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    printf("Prerna Sunil Jadhav - 60004220127\n");
    int arr[] = { 2, 3, 4, 10, 40 };
```

```
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

Output:

```
Prerna Sunil Jadhav - 60004220127
Element is present at index 3
```

Conclusion:

- we can perform all the queries one by one on the sorted array. And since, each binary search query will only take **O(log_{2}N)$O(log_2N)$** time in the worst case, for Q queries it will take ***O(Q**log_{2}Nlog_2N)**.
- So if we use a good sorting algorithm like Merge Sort or Heap Sort which has a guaranteed time complexity of ***O(N**log_{2}Nlog_2N)**, the total time complexity will be
- Binary Search = ***O(N**log_{2}Nlog_2N + Q**log_{2}Nlog_2N) = O((N+Q) * log_{2}Nlog_2N)****
- If we assume the number of queries to be equivalent to the number of elements in the array, then the above equation becomes
- Binary Search = **O(N * log_{2}Nlog_2N)**
- On the other hand, if we use a linear search, then each query will take O(N) in the worst case and the total time complexity for Q queries will be
- Linear Search = **O(N * N) = O(N^{2}N^2)**
- As we can see, binary search performs a lot better than linear search if the array is not sorted and the number of queries is huge, which is the more likely scenario in real-life use cases.
- Although both the Binary search and the Fibonacci search are comparison-based searching methods that use Dynamic Programming, there are many subtle differences between them.
- On average, Fibonacci Search uses 4% more comparisons than Binary search. Binary Search uses division operation (/) to divide range whereas Fibonacci Search doesn't use / albeit, it uses + and -.
- Division and multiplication are costly operations as compared to addition and subtraction. Fibonacci Search reduces the search space by either 2/3 or 1/3. On the other hand, Binary search always shrinks the search space by ½. Furthermore, the Fibonacci search uses 44% more lookups in comparison to the Binary search algorithm.