Jump Search Algorithm

Jump Search Algorithm is a relatively new algorithm for searching an element in a sorted array.

The fundamental idea behind this searching technique is to search fewer number of elements compared to linear search algorithm (which scans every element in the array to check if it matches with the element being searched or not). This can be done by skipping some fixed number of array elements or **jumping ahead by fixed number of steps** in every iteration.

Lets consider a sorted array $A[]$ of size n, with indexing ranging between **0** and **n-1**, and element **x** that needs to be searched in the array $A[]$. For implementing this algorithm, a block of size **m** is also required, that can be skipped or jumped in every iteration. Thus, the algorithm works as follows:

- **Iteration 1**: if ($x==A[0]$), then success, else, if ($x > A[0]$), then jump to the next block.
- **Iteration 2**: if ($x==A[m]$), then success, else, if ($x > A[m]$), then jump to the next block.
- **Iteration 3**: if ($x==A[2m]$), then success, else, if ($x > A[2m]$), then jump to the next block.
- At any point in time, if ($x < A[km]$), then a **linear search** is performed from index $A[(k-1)m]$ to $A[km]$

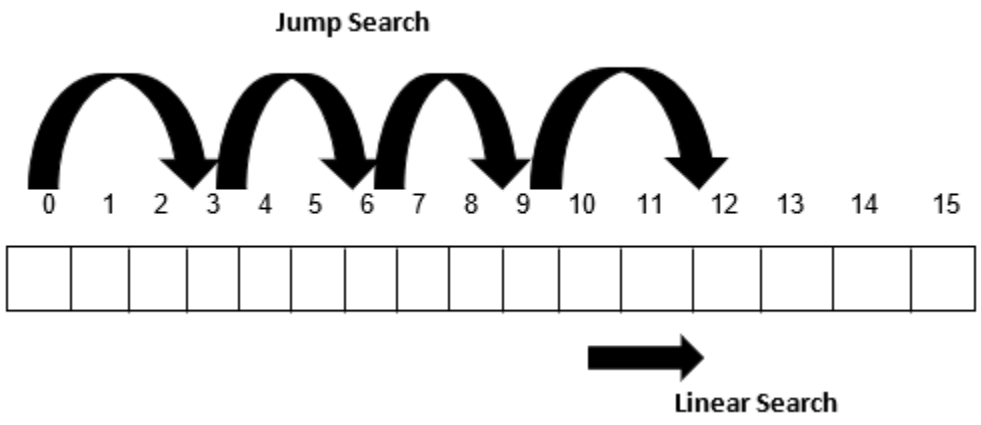

Figure 1: Jump Search technique

Optimal Size of m (Block size to be skipped)

The worst-case scenario requires:

- n/m jumps, and
- $(m-1)$ comparisons (in case of linear search if $x < A[km]$)

Hence, the total number of comparisons will be $(n/m+(m-1))$. This expression has to be minimum, so that we get the smallest value of m (block size).

On differentiating this expression with respect to m and equating it with **0**, we get:

```
n/-m2+1 = 0



n/m2 = 1



m = √n
```

Copy

Hence, the **optimal jump size** is **√n**, where **n** is the size of the array to be searched or the total number of elements to be searched.

Algorithm of Jump Search

Below we have the algorithm for implementing Jump search:

Input will be:

- Sorted array `A` of size `n`
- Element to be searched, say `item`

Output will be:

- A valid location of `item` in the array `A`

Steps for Jump Search Algorithms:

Step 1: Set `i=0` and `m = √n`.

Step 2: Compare `A[i]` with `item`. If `A[i] != item` and `A[i] < item`, then jump to the next block. Also, do the following:

1. Set **i = m**
2. Increment `m` by `√n`

Step 3: Repeat the step 2 till **m < n-1**

Step 4: If `A[i] > item`, then move to the beginning of the current block and perform a linear search.

1. Set **x = i**
2. Compare `A[x]` with `item`. If `A[x]== item`, then print `x` as the valid location else set `x++`
3. Repeat Step 4.1 and 4.2 till **x < m**

Step 5: Exit

Pictorial Representation of Jump Search with an Example

Let us trace the above algorithm using an example:

Consider the following inputs:

- `A[]` = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
- `item` = 77

Step 1: **m = √n = 4** (Block Size)

Step 2: Compare `A[0]` with `item`. Since **A[0] != item** and **A[0]<item**, skip to the next block

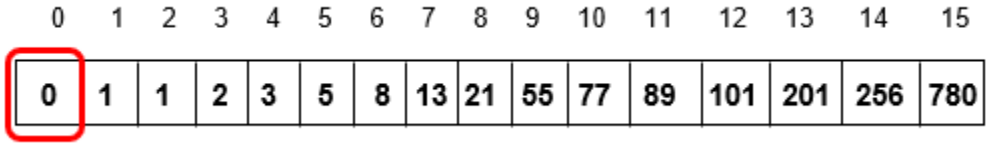

Figure 2: Comparing A[0] and item

Step 3: Compare A[3] with item. Since A[3] != itemand A[3]<item, skip to the next block

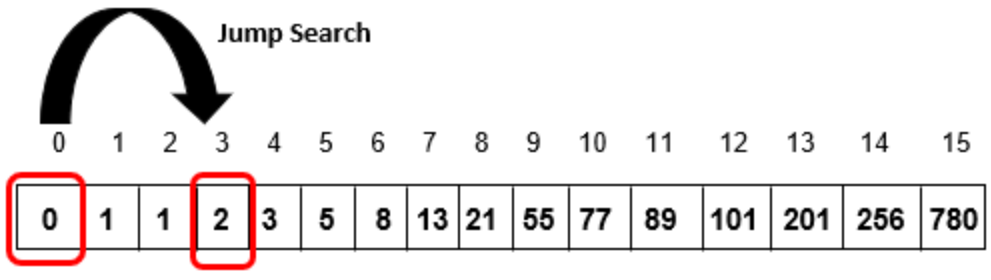

Figure 3: Comparing A[3] and item

Step 4: Compare A[6] with item. Since A[6] != itemand A[6]<item, skip to the next block

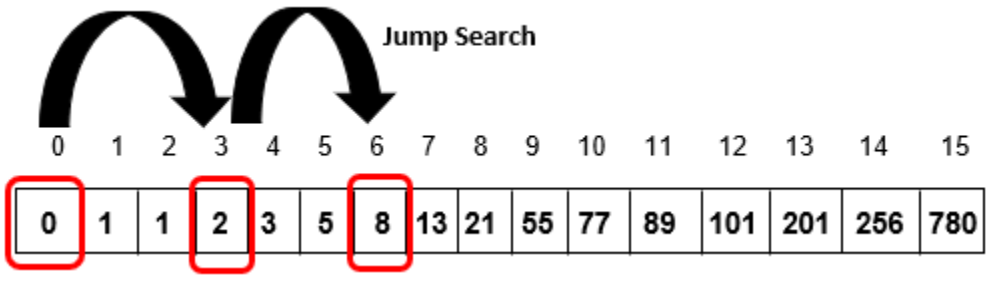

Figure 4: Comparing A[6] and item

Step 5: Compare A[9] with item. Since A[9] != itemand A[9]<item, skip to the next block

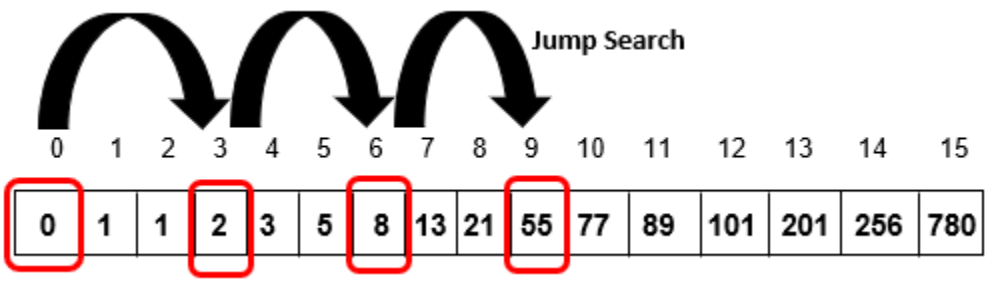

Figure 5: Comparing A[9] and item

Step 6: Compare A[12] with item. Since A[12] != item and A[12] >item, skip to A[9] (beginning of the current block) and perform a linear search.

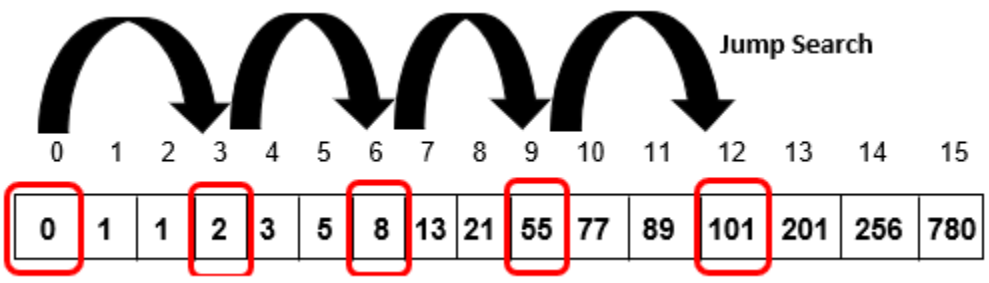

Figure 6: Comparing A[12] and item

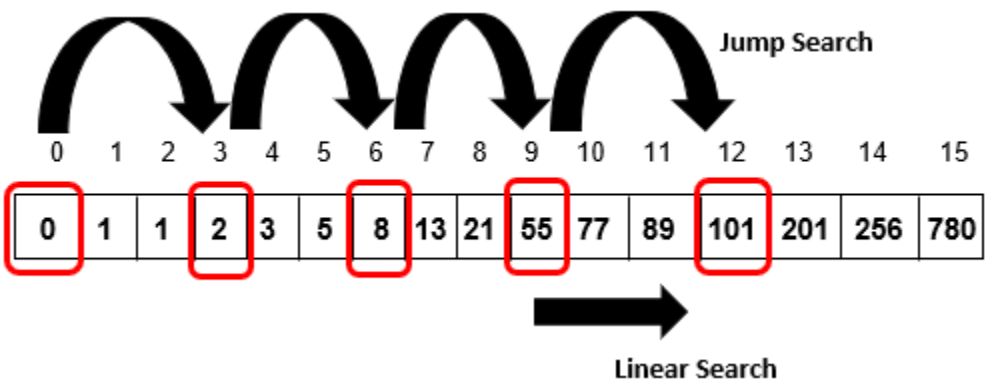

Figure 7: Comparing A[9] and item (Linear Search)

- Compare A[9] with item. Since A[9] != item, scan the next element
- Compare A[10] with item. Since A[10] == item, index 10 is printed as the valid location and the algorithm will terminate

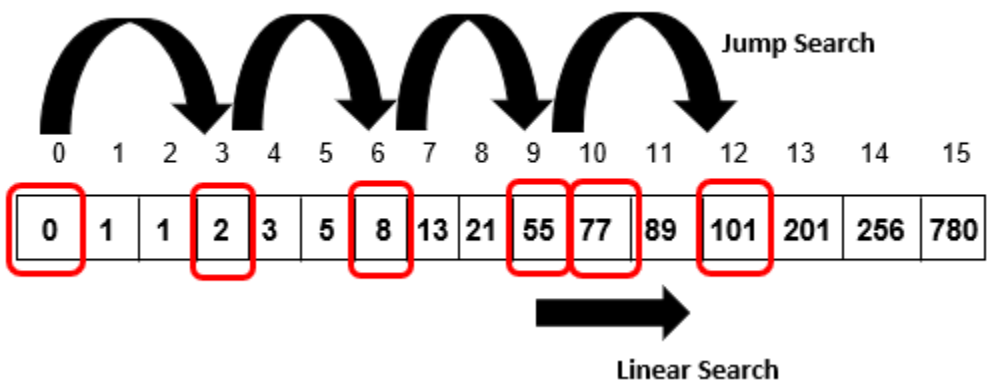

Figure 8: Comparing A[10] and item (Linear Search)

Implementation of Jump Search Algorithm

Following is the program in which we have implemented the Jump search algorithm in C++ language:

```
#include<iostream>

#include<cmath>



using namespace std;

int jump_Search(int a[], int n, int item) {

   int i = 0;

int m = sqrt(n); //initializing block size= √(n)


while(a[m] <= item && m < n) {

   // the control will continue to jump the blocks

   i = m;  // shift the block

   m += sqrt(n);

   if(m > n - 1)  // if m exceeds the array size

      return -1;

}


for(int x = i; x<m; x++) { //linear search in current block

   if(a[x] == item)

      return x; //position of element being searched

   }

   return -1;

}


int main() {

   int n, item, loc;

   cout << "\n Enter number of items: ";

   cin >> n;
```

```
   int arr[n]; //creating an array of size n

   cout << "\n Enter items: ";


   for(int i = 0; i< n; i++) {

      cin >> arr[i];

   }


   cout << "\n Enter search key to be found in the array: ";

   cin >> item;

   loc = jump_Search(arr, n, item);

   if(loc>=0)

      cout << "\n Item found at location: " << loc;

   else

      cout << "\n Item is not found in the list.";

}
```

Copy

The input array is the same as that used in the example:

```
Enter number of items: 16

Enter items: 0 1 1 2 3 5 8 13 21 55 77 89 101 201 256 780

Enter search key to be found in the array: 77

Item found at location: 10
```

Note: The algorithm can be implemented in any programming language as per the requirement.

Complexity Analysis for Jump Search

Let's see what will be the time and space complexity for the Jump search algorithm:

Time Complexity:

The while loop in the above C++ code executes n/m times because the loop counter increments by m times in every iteration. Since the optimal value of m= √n , thus, n/m=√n resulting in a time complexity of **O(√n)**.

Space Complexity:

The space complexity of this algorithm is **O(1)** since it does not requireany other data structure for its implementation.

Key Points to remember about Jump Search Algorithm

- This algorithm works only for sorted input arrays
- Optimal size of the block to be skipped is √n, thus resulting in the time complexity O(√n^2)
- The time complexity of this algorithm lies in between linear search (O(n)) and binary search (O(log n))
- It is also called block search algorithm

Advantages of Jump Search Algorithm

- It is faster than the linear search technique which has a time complexity of O(n) for searching an element

Disadvantages of Jump Search Algorithm

- It is slower than binary search algorithm which searches an element in O(log n)
- It requires the input array to be sorted
