**Jump Search**

# Jump Search

A jump search or block search refers to a search algorithm for ordered lists. It works by first checking all items *Lkm*, where k ∈ N {\displaystyle k\in \mathbb {N} }and *m* is the block size, until an item is found that is larger than the search key. To find the exact position of the search key in the list a linear search is performed on the sublist *L*[(*k*-1)*m*,*km*].

The optimal value of *m* is √*n*, where *n* is the length of the list *L*. Because both steps of the algorithm look at, at most, √*n* items the algorithm runs in O(√*n*) time.

This is better than a linear search, but worse than a binary search. The **advantage** over the latter is that a jump search only needs to jump backwards once, while a binary can jump backwards up to log n times. This can be important if a jumping backwards takes significantly more time than jumping forward.

The algorithm can be modified by performing multiple levels of jump search on the sublists, before finally performing the linear search. For an *k*-level jump search the optimum block size *ml* for the *l*th level (counting from 1) is *n*(k-l)/k. The modified algorithm will perform *k* backward jumps and runs in O(*kn*1/(*k*+1)) time.

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.

# Basic Algorithm

# Pseudocode

Input: An ordered list L, its length n and a search key s

Output: The position of s in L, or nothing if s is not in L

a ← 0

b ← ⌊√n⌋

while Lmin(b,n)-1 < s do

a ← b

b ← b + ⌊√n⌋

if a ≥ n then

return nothing

while La < s do

a ← a + 1

if a = min(b,n)

return nothing

if La = s then

return a

else

return nothing

# C/C++ Implementation

int jumpSearch(int arr[], int x, int n)

{

    // Finding block size to be jumped

    int step = sqrt(n);

    // Finding the block where element is present (if it is present)

    int prev = 0;

    while (arr[min(step, n)-1] < x)

    {

        prev = step;

        step += sqrt(n);

        if (prev >= n)

            return -1;

    }

    // Doing a linear search for x in block beginning with prev.

    while (arr[prev] < x)

    {

        prev++;

        // If we reached next block or end of array, element is not present.

        if (prev == min(step, n))

            return -1;

    }

    // If element is found

    if (arr[prev] == x)

        return prev;

    return -1;

}

# Complexity

**Best case**:

**Average case**:

**Worst case**: 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.

**total number of comparisons in the worst case : ((n/m) + m-1)**

**Its value will be min when m = √n.**

\* **Auxiliary Space**: O(1)

1. The optimal size of a block to be jumped is O(√ 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average case** | **Worst case** | **Space Complexity** |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(logn) | O(logn) | O(1)\* |
| Jump Search | O(1) | O(√ n) | O(√ n) | O(1) |
| Interpolation Search | O(1) | O (log log n)) | O(n) | O(1) |
| Exponential Search | O(1) | O(log i) | O(log i) | O(1) |
| Fibonacci Search | O(1) | O(logn) | O(logn) | O(1) |

# Application

This can be important if a jumping backwards takes significantly more time than jumping forward.

# Example

#include <bits/stdc++.h>

using namespace std;

int jumpSearch(int arr[], int x, int n)

{

    // Finding block size to be jumped

    int step = sqrt(n);

    // Finding the block where element is present (if it is present)

    int prev = 0;

    while (arr[min(step, n)-1] < x)

    {

        prev = step;

        step += sqrt(n);

        if (prev >= n)

            return -1;

    }

    // Doing a linear search for x in block beginning with prev.

    while (arr[prev] < x)

    {

        prev++;

        // If we reached next block or end of array, element is not present.

        if (prev == min(step, n))

            return -1;

    }

    // If element is found

    if (arr[prev] == x)

        return prev;

    return -1;

}

// Driver program to test function

int main()

{

    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,

                 34, 55, 89, 144, 233, 377, 610 };

    int x = 55;

    int n = sizeof(arr) / sizeof(arr[0]);

    // Find the index of 'x' using Jump Search

    int index = jumpSearch(arr, x, n);

    // Print the index where 'x' is located

    cout << "\nNumber " << x << " is at index " << index;

    return 0;

}

Output:

Number 55 is at index 10

# References

<https://www.geeksforgeeks.org/searching-algorithms/>

<https://www.geeksforgeeks.org/jump-search/>