**Jump search**

**Jump search** (also known as **block search**) is an algorithm for finding the position of an element in a sorted array. Unlike linear search, it doesn't compare each element of an array with the target value. Instead, to find a value, the array is represented as a sequence of blocks.

It's proved that the optimal block size is sqrt(n)**,**where n is the size of the array. In this case, the algorithm performs sqrt(n) + sqrt(n)comparisons in the worst case, so the time complexity is O(sqrt(n)).This is more efficient than the linear search algorithm.

**Principle 1.** If the array is ascending, an element of a block is less than or equal to a value of the following block.

**Principle 2.** If a value is not present in the first block from the beginning whose right border exceeds the value, it is not present in the array at all.

If the right border of a block is greater than the target element, we've found a block that may contain the target value. When the block is found, the algorithm performs a backward linear search within the block. If the target value is found, it returns its index; otherwise, the array does not contain the target.

Further, we will consider the algorithm with the jump size equal to sqrt(n).

Please keep in mind the following:

* If sqrt(n) is not an integer value, we take only the integer part;
* If the index of the following element to jump to is greater than the last element index, we jump to the last element.

It can be recursive.

## Implementation in Java

Let's consider how the algorithm can be implemented in Java. The jumpSearch method finds a block where the target element may be presented and then invokes backwardSearch to search the element in this block.

public static int jumpSearch(int[] array, int target) {  
    int currentRight = 0; // right border of the current block  
    int prevRight = 0; // right border of the previous block  
   
    /\* If array is empty, the element is not found \*/  
    if (array.length == 0) {  
        return -1;  
    }  
   
    /\* Check the first element \*/  
    if (array[currentRight] == target) {  
        return 0;  
    }  
   
    /\* Calculating the jump length over array elements \*/  
    int jumpLength = (int) Math.sqrt(array.length);  
   
    /\* Finding a block where the element may be present \*/  
    while (currentRight < array.length - 1) {  
   
        /\* Calculating the right border of the following block \*/  
        currentRight = Math.min(array.length - 1, currentRight + jumpLength);  
   
        if (array[currentRight] >= target) {  
            break; // Found a block that may contain the target element  
        }  
   
        prevRight = currentRight; // update the previous right block border  
    }  
   
    /\* If the last block is reached and it cannot contain the target value => not found \*/  
    if ((currentRight == array.length - 1) && target > array[currentRight]) {  
        return -1;  
    }  
   
    /\* Doing linear search in the found block \*/  
    return backwardSearch(array, target, prevRight, currentRight);  
}  
   
public static int backwardSearch(int[] array, int target, int leftExcl, int rightIncl) {  
    for (int i = rightIncl; i > leftExcl; i--) {  
        if (array[i] == target) {  
            return i;  
        }  
    }  
    return -1;  
}

This implementation may look a little cumbersome, but it has its advantages:

* if the array is empty, it immediately returns the result (not found);
* if the first element matches the target, it immediately returns the result (found);
* if the target is not found in the block in which it could be present, the algorithm doesn't search in the remaining blocks which cannot contain the target (it relies on the fact that the input array is sorted).

In the presented algorithm, once we have found the block that may contain the target value, we perform the backward linear search. But we could perform another jump search within the block (backward or forward)! And then recursively perform jump search until we are left with only one element.

This version will perform \sqrt n + \sqrt[4] n + \sqrt[8] n + ... + 1*n*​+4*n*​+8*n*​+...+1 comparisons in the worst case. It's faster than the base implementation but is still O(\sqrt n)*O*(*n*​).