Theory: Jump search

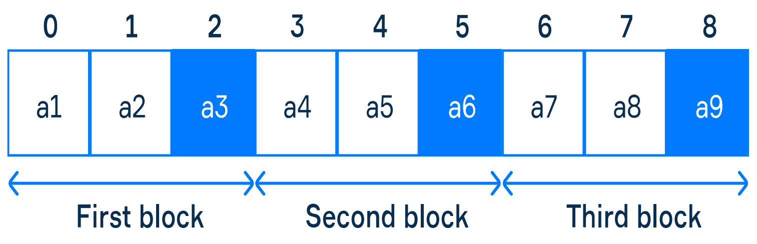
13 minutes 6 / 6 problems solved

Start practicing

1555 users solved this topic. Latest completion was about 8 hours ago.

§1. Introduction

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



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 algorithm is more efficient than the linear search algorithm.

Current topic:

✓ Jump search

Topic depends on:

✓ Linear search

Topic is required for:

Jump search in Java

§2. Basic Principles

Consider the basic principles of this algorithm for searching in ascending sorted arrays. Note that it can search in descending sorted arrays as well.

Principle 1. For ascending sorted arrays, any value from a block is less than or equal to any value from the following block.

Principle 2. If the target element is not present at the beginning of the first block, and its right border exceeds the target element, it is not present in the array at all.

The algorithm jumps over blocks to find the block that may contain the target element. So, the algorithm compares the right borders of blocks to the target element.

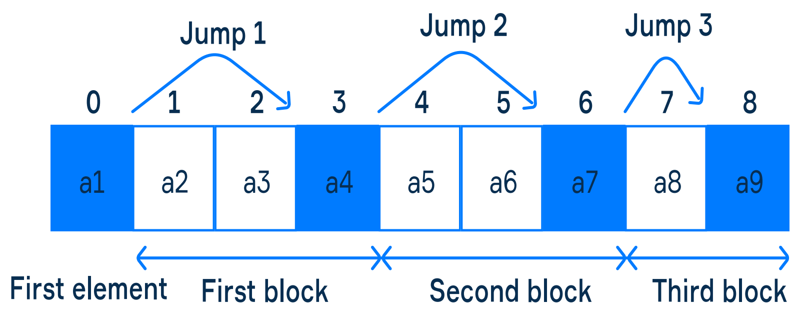
Table of contents:

§3. Searching

If the right border of a block is equal to the target element, we have found it. Sometimes we need to search the target with the minimum index.

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

Sometimes blocks don't include the first array element, and then the algorithm works in the same way as described above. The complexity of the algorithm doesn't change.



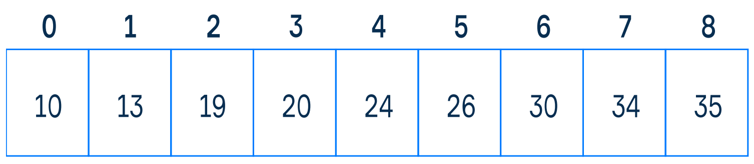
Further, we will consider the algorithm with the jump size equal to $\sqrt(n)$.

Please keep in mind the following:

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

§4. Example

Suppose we have a sorted ascending array of nine integers:



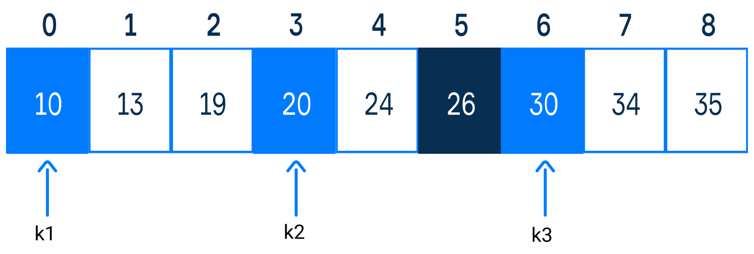
The input array has nine elements with indices 0 to 8. We want to find the index of the value 26 using jump search.

Our first step would be finding the block that may contain the target value. The jump length is $\sqrt{(9)} = 3$.

1) The first element (10) is less than the target value (26), so we jump to the next element with the index $0 + 3 = 3$.

2) The element with the index 3 (20) is less than the target value (26); we jump to the next element with the index $3 + 3 = 6$.

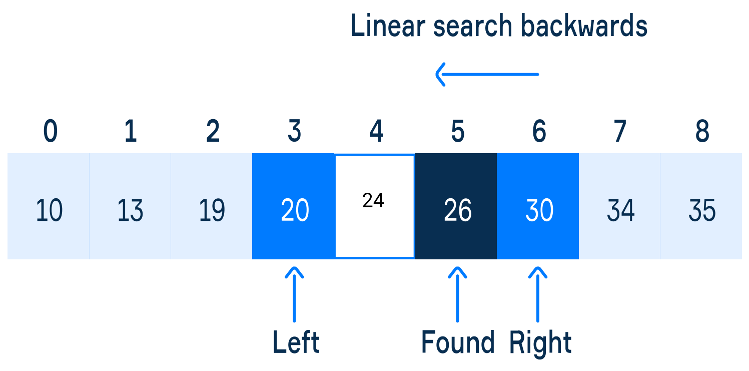
3) The element with the index 6 (30) is greater than the target value (26).



During the stage, we store indices of the current and the previously considered element to use them in the second stage.

Second, we do a **backward linear search.**

We have the left and the right indices of the block that may contain the target value. The left is *k2,* the right is *k3*. Now we will consider only the elements belonging to this block.



After doing a linear search between these indices, we have found our element (26) and its index 5.

So, the idea is that the algorithm finds the blocks where the target element may be present and then uses linear search within the block.

§5. Harder, faster, stronger

In this algorithm, once we find the block that may contain the value, we perform a backward linear search. However, what we could also do perform another jump search within the block (backward or forward) and then recursively perform jump search until we have only one element.

This version will perform $\sqrt{(n)} + \sqrt{(\sqrt{(n)})} + \sqrt{(\sqrt{(\sqrt{(n)})})} + \ldots + 1$ comparisons in the worst case. It's faster than the base implementation, but is still $O(\sqrt{(n)})$.

If you're feeling up to the challenge, you may try to toy with this algorithm.

Report a typo

147 users liked this theory. 9 didn't like it. **What about you?**

Start practicing

Comments (13) Hints (0) Useful links (0) Show discussion