Theory: Jump search in Java

56 minutes 0 / 4 problems solved

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. It splits the given array into a sequence of blocks and then jumps over them to find a block that may contain the target element. To do that, the algorithm compares the right borders of blocks with the target element.

If n is the size of a block, the algorithm performs $\sqrt{n} + \sqrt{n}$ comparisons in the worst case. It means the time complexity is $O(\sqrt{n})$ that is more efficient than **linear search.**

§1. 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.

Current topic:

Jump search in Java

Topic depends on:

✓ Jump search

✗ Algorithms in Java

Table of contents:

↑ Jump search in Java

§1. Implementation in Java

§2. Usage examples

§3. A faster jump search algorithm

Feedback & Comments

```
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 rightI
ncl) {
```

```
    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).

§2. Usage examples

Let's see an example of how the method works:

```
int[] array = { 10, 13, 19, 20, 24, 26, 30, 34, 35 };

jumpSearch(array, 10); // 0
jumpSearch(array, 13); // 1
jumpSearch(array, 19); // 2
jumpSearch(array, 20); // 3
jumpSearch(array, 24); // 4
jumpSearch(array, 26); // 5
jumpSearch(array, 30); // 6

jumpSearch(array, 34); // 7

jumpSearch(array, 35); // 8



jumpSearch(array, -10); // -1

jumpSearch(array, 11);  // -1

jumpSearch(array, 27);  // -1

jumpSearch(array, 37);  // -1
```

So, we found all the elements of the array and did not find the missing ones.

If you do not quite understand the algorithm, run these tests in the debug mode.

§3. A faster jump search algorithm

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 $\sqrt{n} + \sqrt[4]{n} + \sqrt[8]{n} + ... + 1$ comparisons in the worst case. It's faster than the base implementation but is still $O(\sqrt{n})$.

You can try to implement this algorithm if you'd like.

Report a typo

68 users liked this theory. **5** didn't like it. What about you?

Start practicing

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