Jump search algorithm

Source: ***https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms***

1. *What is a Jump search algorithm?*

Jump search is an efficient search algorithm for finding an element in a sorted array. It works by first comparing the element to be searched with the element at the midpoint of the array. If the element to be searched is smaller than the midpoint element, then it must be present in the left subarray. If the element to be searched is greater than the midpoint element, then it must be present in the right subarray.

The jump search algorithm works by jumping through the array in fixed steps, called the block size, until it finds the element or determines that it is not present in the array. The block size is chosen such that the time complexity of the algorithm is $O(\sqrt{n})$.

2. *Code example.*

```
#include <iostream>
#include <cmath>

int jumpSearch(int arr[], int n, int x) {
  // Calculate the block size to be jumped
  int blockSize = sqrt(n);
  // Find the block where the element is present
  int prev = 0;
  while (arr[std::min(blockSize, n)-1] < x) {
    prev = blockSize;
    blockSize += sqrt(n);
    if (prev >= n) {
      return -1;
    }
  }
  // Do 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 == std::min(blockSize, n)) {
      return -1;
    }
  }
  // If element is found
  if (arr[prev] == x) {
    return prev;
  }
  return -1;
}
```

```
int main() {
  int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
377, 610 };
  int n = sizeof(arr) / sizeof(arr[0]);
  int x = 55;
  // Find the index of x using Jump Search
  int index = jumpSearch(arr, n, x);
  // Print the index where x is located
  std::cout << "Number " << x << " is at index " << index <<
std::endl;
  return 0;
}
```

```
Output: Number 55 is at index 10
```

Element **55** is located at index **10** in the array. The jumpSearch function returns **10** as the index at which the element is located, and this value is printed by the main function.

3. Code explanation.

The jumpSearch function takes in an array arr of integers, the size of the array n, and the element to be searched x. It first calculates the block size to be jumped using the formula:

```
blockSize = sqrt(n)
```

The block size is chosen such that the time complexity of the algorithm is **O(√n).** This means that the time taken to search for an element in the array increases as the square root of the size of the array.

Next, the function finds the block where the element is likely to be present by iterating through the array in blocks of size **blockSize** until it finds a block where the last element is greater than or equal to the element to be searched x. The **prev** variable is used to keep track of the starting index of the current block.

```
int prev = 0;
while (arr[std::min(blockSize, n)-1] < x) {
  prev = blockSize;
  blockSize += sqrt(n);
  if (prev >= n) {
    return -1;
  }
}
```

If the element is not found in any of the blocks, the function returns **-1**.

Once the block where the element is likely to be present is found, the function does a linear search within that block to find the element.

```
  while (arr[prev] < x) {
    prev++;
    // If we reached next block or end of array, element is not present
    if (prev == std::min(blockSize, n)) {
      return -1;
    }
  }
  // If element is found
  if (arr[prev] == x) {
    return prev;
  }
```

If the element is found, the function returns the index at which it is located. If the element is not found, the function returns **-1**.

Finally, the main function calls the jumpSearch function to search for the element **55** in the sorted array arr. If the element is found, the index at which it is located is printed.

4. Time complexity.

The time complexity of jump search is **O(√n).**

This means that the time taken to search for an element in the array increases as the square root of the size of the array.

In the jumpSearch function, the block size is chosen such that the time complexity of the algorithm is **O(√n).** The function first jumps through the array in blocks of size **blockSize**, and then does a linear search within the block where the element is likely to be present.

The time complexity of the algorithm can be expressed as:

$$T(n) = O(\sqrt{n}) + O(\sqrt{n})$$
$$= O(\sqrt{n})$$

The first term, **O(√n),** represents the time taken to jump through the array in blocks. The second term, **O(√n),** represents the time taken to do the linear search within the block.

I wish you happy learning,

Your Rosen Rusev