Interpolation search algorithm

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

1. What is interpolation search algorithm?

Interpolation search is an algorithm for searching for a specific value in a sorted list of values. It is an improvement over the binary search algorithm, which has a worst-case complexity of **`O(log n).`** Interpolation search has a worst-case complexity of **`O(log log n),`** which is faster than binary search for large lists.

The interpolation search algorithm works by using the sorted nature of the list to estimate the position of the target value. It starts by calculating the position of the target value using a formula that takes into account the value of the target, the minimum and maximum values in the list, and the length of the list. It then checks the value at this estimated position to see if it is the target value. If it is not, the algorithm determines whether the target value is higher or lower than the value at the estimated position, and adjusts the estimated position accordingly. This process is repeated until the target value is found or it is determined that the target value is not present in the list.

One key advantage of interpolation search is that it can be more efficient than binary search for lists where the values are uniformly distributed. This is because interpolation search is able to make better use of the sorted nature of the list to estimate the position of the target value, whereas binary search always divides the list in half. However, interpolation search can be slower than binary search for lists where the values are not uniformly distributed, since the estimates of the position of the target value may be less accurate in these cases.

2. Code example.

```
#include <algorithm>

#include <iostream>

#include <vector>

int interpolationSearch(const std::vector<int>& list, int target) {

  int left = 0;

  int right = list.size() - 1;

  while (left <= right) {

    // Calculate the position of the target value using the interpolation formula

    int pos = left + ((target - list[left]) * (right - left)) / (list[right] - list[left]);

    // Check the value at this position

    if (list[pos] == target) {

      // The target value has been found

      return pos;

    } else if (list[pos] < target) {

      // The target value is higher than the value at this position, so search the right half of
the list

      left = pos + 1;

    } else {

      // The target value is lower than the value at this position, so search the left half of
the list

      right = pos - 1;

    }

  }

  // The target value was not found in the list

  return -1;

}

int main() {

  std::vector<int> list{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  int target = 5;

  int index = interpolationSearch(list, target);

  if (index == -1) {

    std::cout << "The target value was not found in the list." << std::endl;

  } else {

    std::cout << "The target value was found at index " << index << "." << std::endl;

  }

  return 0;

}
```

Output: `The target value was found at index 4.`

This is because the target value (**5**) is present in the list at index **4**, and the **`interpolationSearch`** function correctly returns this index.

It's worth noting that the output of the **`interpolationSearch`** function may vary depending on the input list and target value. If the target value is not present in the list, the function will return -1. If the target value is present in the list multiple times, the function will return the index of the first occurrence of the target value.

3. *Code explanation.*

The **`interpolationSearch`** function takes a sorted list of integers (represented as a std::vector) and a target value as input. It defines two variables, left and right, which represent the indices of the left and right ends of the sublist being searched. Initially, left is set to 0 (the first index in the list) and right is set to the last index in the list (the size of the list minus 1).

The function then enters a while loop, which continues until left is greater than right. Inside the while loop, the function calculates the estimated position of the target value using the following formula:

```
int pos = left + ((target - list[left]) * (right - left)) /
(list[right] - list[left]);
```

This formula takes into account the value of the target, the minimum and maximum values in the sublist being searched (**`list[left]`** and **`list[right]`**), and the length of the sublist (right - left). It uses these values to estimate the position of the target value within the sublist.

Next, the function checks the value at the estimated position using an if statement. If the value at this position is equal to the target value, the function returns the index of the target value (**`pos`**).

If the value at the estimated position is not equal to the target value, the function uses another if statement to determine whether the target value is higher or lower than the value at the estimated position. If the target value is higher, the function sets left to pos + 1 and searches the right half of the list. If the target value is lower, the function sets right to pos - 1 and searches the left half of the list.

If the while loop ends and the target value has not been found, the function returns -1 to indicate that the target value is not present in the list.

Finally, the main function creates a sorted list of integers and calls the **`interpolationSearch`** function with this list and a target value as input. If the target value is found, the function prints the index at which it was found. If the target value is not found, the function prints a message indicating that the target value was not found in the list.

4. Time complexity.

The time complexity of the **`interpolationSearch`** function in the code example above is **`O(log log n).`**

This means that the function's runtime increases at a slower rate as the size of the input list (**`n`**) increases. Specifically, the function's runtime increases logarithmically with the logarithm of **`n`**, rather than linearly as in the case of a linear search or logarithmically as in the case of a binary search.

The time complexity of **`O(log log n)`** makes interpolation search faster than binary search for large lists, but it may not be as efficient as other search algorithms for smaller lists or lists where the values are not uniformly distributed.

I wish you happy learning,

Your Rosen Rusev