Searching Algorithms

Introduction

Searching is one of the most important topics in computing, in this case, we are going to work with the basic types of searching, Linear, and Binary.

Linear Search

Introduction

Consist in search an element in the array looking for each index in the array and comparing the value.

This algorithm has a computing complex of $O(n) = n$.

```
function linear_search(A, n, Key)
  index := 0
  while index < n do
    if A[index] == Key then
      return index
    index = index + 1
  return -1
```

Proof Complex Time is O(n).

$$T(n) = 1 + n(2) = 2n + 1 \longrightarrow O(n) = n$$

Implementation

```
public static int Search(int[] array, int key)
{
    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] == key)
        {
            return i;
        }
    }
    return -1;
}
```

Binary Search

Introduction

Is a Better technique than the linear search, the binary search satisfies the next points.

- Array in sorted order.
- Examine the middle element.
- If matches, return the index.

- If key < middle element, search lower half
- If key > middle element, search upper half.

Binary Search has two forms, iterative and recursive form.

Iterative algorithm

```
function binary_iterative(A, n, key)
  L := 0
  R := n-1
  while L <= R do
    m := floor((L+R)/2)
    if key == A[m] then
      return m
    else if key < A[m] then
      R := m - 1
    else if key > A[m] then
      L := m + 1
  return -1
```

The complexity Time is the next.

In the first part we have O(1) + O(1), in the assign of L and R. In the while, we have that in each iteration L or R is decreasing or increasing n/2, i.e. the while loop takes $log_2(n)$.

$$T(n) = 1 + 1 + log_2(n) \cdot (1 + 1 + 1 + 1 + 1)$$
$$T(n) = 2 + 5 \cdot log_2(n) \longrightarrow O(n) = log(n)$$

Recursive algorithm

```
function binary_recursive(A, key, L, R)
  if L > R then
    return -1
  else
    m := floor( (L+R)/2 )
    if key == A[m] then
      return m
    else if key < A[m] then
      return binary_recursive(A, key, L, m-1)
    else if key > A[m] then
      return binary_recursive(A, key, m + 1, R )
```

The complexity Time is the next.

First, we are going to take a look at the compartment of the time complexity.

In this case, n is the length of array $T(n) = T(n/2) + 1$ Because we are creating a half and comparing.

Now we can compare each case without loss of generality, supposing that $n = 2^k, k \in \mathbb{N}$.

In the case $T(1) = 0$, because with one item we can find the right one without comparison.

Similarly in the case $T(2) = 1$, $T(4) = 2$, and so on. Each case satisfies $T(n) = log_2(n)$.

PROOF

Proof no loss of generality $T(2^n) = log_2(2^n) = n$.

By Induction over n, let $n \in \mathbb{N}$, let's see the base case.

$T(2^1) = 1 = log_2(1) \longrightarrow T(2^1) = log_2(1)$, it's true for $n = 0$

Let $k \in \mathbb{N}$ be given and suppose $T(2^n)$ is true for $n = k$, then let's see what happens with $k + 1$.

$T(2^{k+1}) = T((2^k \cdot 2^1)/2) + 1$ By definition.

$T(2^{k+1}) = T(2^k) + 1$ By inductive hypothesis.

$T(2^{k+1}) = k + 1$, it's true for every $k \in \mathbb{N}$.

So, using induction we can prove that $T(n) = log_2(n)$

ANOTHER PROOF USING "Master Theorem".

Using the Master Theorem, $T(n) = aT(n/b) + f(n).$ we have $a = 1, b = 2$ and $f(n) = 1 = c$, where c is a constant. In this case, the key in the Master Theorem is $log_b(a) = log_2(1) = 0$, Here we are in case 2 since by taking $k = 0$ we find that $n^{log_b(a)}(log(n))^k = (n^0)(log(n))^0 = 1$ therefore, $f(n) = c = \Theta(n^{log_b a} log^k n)$

From Case 2 of the Master Theorem we know that $T(n) = \Theta(n^{log_b a}(log(n))^{k+1})$ which in this case yields $T(n) = \Theta(n^0(log(n))^1) = \Theta(log(n))$

Implementation

```
public int BinarySearchIterative(int[] array, int n, int key)
{
    int low = 0;
    int high = n - 1;

    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (key == array[mid])
        {
            return mid;
        }
        if (key < array[mid])
        {
            high = mid - 1;
        }
        else
        {
            low = mid + 1;
        }
    }

    return -1;
}
```

```
public int BinarySearchRecursive(int[] array, int key, int low, int high)
{
    if (low > high)
    {
        return -1;
    }

    int mid = (low + high) / 2;
    if (key == array[mid])
    {
        return mid;
    }
    if (key < array[mid])
    {
        return BinarySearchRecursive(array, key, low, mid - 1);
    }
    else
    {
        return BinarySearchRecursive(array, key, mid + 1, high);
    }
}
```