BINARY SEARCH

The binary search, also known as **half-interval search**, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array.

Binary search runs in logarithmic time in the worst case, making O(log n) comparisons. It takes constant O(1) space in the array. Binary search is faster than linear search except for small arrays.

Binary search works on sorted arrays. Binary search begins by comparing the middle element of the array with the target value.

If the target value matches the middle element, its position in the array is returned.

If the target value is less than the middle element, the search continues in the lower half of the array.

If the target value is greater than the middle element, the search continues in the upper half of the array.

By doing this, the algorithm eliminates the half in which the target value cannot lie in each iteration.

Procedure Binary_Search(A, Key)

Given an array A of n elements with values or records A_1, A_2, … , A_N sorted such that $A_1 \le A_2 \le \ldots \le A_N$, and search value Key, this procedure finds the index of Key in A. LOW, HIGH and MID are local index variables.

```
1. Initialize variables
    LOW ← 1
    HIGH ← N
2. Set up the iteration
    Repeat thru Step 4 While LOW <= HIGH
3. Compute the Middle Index
    MID ← (HIGH + LOW)/2             ... Integer-Division
4. Search the Key
    If A(MID) < Key Then
        LOW ← MID + 1                ... Search in Left Half
    IF A(MID) > Key Then
        HIGH ← MID - 1               ... Search in Right Half
    Return MID
```

LINEAR SEARCH

The linear search or sequential search is a method for finding an element within a list.

A linear search runs in at worst linear time and makes at most n comparisons. If each element is equally likely to be searched, then linear search has an average case of n/2 comparisons, but the average case can be affected if the search probabilities for each element vary.

A linear search sequentially checks each element of the list until it finds an element that matches the target value. If the algorithm reaches the end of the list, the search terminates unsuccessfully.

Procedure Linear_Search (A, Key)

Given an array A of n elements with values or records A1, A2 , … , An, and search value "Key", this procedure finds the index of "Key" in A. NDX is a local index variable.

```
    1. Initialize variables
        NDX ← 1
    2. Set up the iteration
        Repeat thru Step 4 While NDX <= n
    3. Search the Key
        If A(NDX) = Key Then
            Return NDX                  ... Search Successful
    4. Increment to next array element
        NDX ← NDX + 1
    5. Report Unsuccessful Search
        Return 0                        ... Search Unsuccessful
```

COUNTING SORT

The counting sort, and its application to radix sorting, were both invented by Harold H. Seward in 1954.

The Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers.

It is an **integer sorting algorithm**. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence.

Its running time is **linear** in the number of items and the difference between the maximum and minimum key values, hence only suitable in situations where the variation in keys is not significantly greater than the number of items.

It is used as a subroutine in radix sort that can handle larger keys more efficiently.

Because counting sort uses **key values** as indexes into an array, it is not a comparison sort, and the $\Omega(n \log n)$ lower bound for comparison sorting does not apply to it.

The input to counting sort consists of a collection of **n** items, each of which has a non-negative integer key whose maximum value is at most **k**.

The output is an array of the items, in order by their keys.

Counting sort is a stable sort but isn’t an in-place algorithm. Other than the count array, it needs separate input and output arrays.

Procedure Counting_Sort(A, K)

Given an input array A, be sorted, and K is the range of non-negative key values this procedure sorts the array in ascending order of keys. KOUNT[] is intermediate array to store histogram of keys in the defined range K. SORTED[] is the output array.

```
1. Initialize local variables.
    KNT ← 1
2. Initialize KOUNT[] array
    Repeat While KNT <= K
        KOUNT(KNT) ← 0
3. Create the histogram
    Repeat For KNT = 1 to A.Length
        KOUNT(A(KNT)) ← KOUNT(A(KNT)) + 1
4. Normalize the KOUNT[] array
    Repeat For KNT = 2 to K
        KOUNT(KNT) ← KOUNT(KNT) + KOUNT(KNT-1)
```

```
5. Order the keys
    Repeat For KNT = A.Length downto 1
        SORTED(KOUNT(A(KNT))) := A(KNT)
        KOUNT(A(KNT)) := KOUNT(A(KNT)) - 1
6. Return the sorted list
    Return SORTED[]
```

Complexity Analysis

====================

- The initialization of the count array take O(k) time.
- The initialization of the output array take O(n) time.
- The algorithm takes O(n + k) steps.