left half of the array. If the key is greater than the middle element, apply the same algorithm for the right half of the array. The algorithm for the binary search is

```
intbinary_search(intA[], intkey, int min, int max)
{
int mid = [(min + max)/2];

if (A[mid] > key)
    returnbinary_search(A, key, min, mid − 1);
else if (A[mid] < key)
    returnbinary_search(A, key, mid + 1, max);
else
    / key has been found
    return mid;
}
```

$int binary_s earch(int A[], int key, int \min, int \max)$

{

int mid $= [(\min + \max)/2]$;

if $(A[mid] > key)$

$return binary_s earch(A, key, \min, mid - 1)$;

else if $(A[mid] < key)$

$return binary_s earch(A, key, mid + 1, \max)$;

else

/ key has been found

return mid;

}

Calculation of the $mid - element$ takes a unit time. If the key does not match with the $mid - element$, the search is limited to $n/2$ space. Hence, the time taken by the algorithm is

$$T(n) = T(\frac{n}{2}) + 1$$
$$T(\frac{n}{2}) = T(\frac{n}{2^2}) + 1$$
$$T(\frac{n}{2^2}) = T(\frac{n}{2^3}) + 1$$
$$\ldots\ldots$$
$$\ldots\ldots$$
$$T(\frac{n}{2^{i-1}}) = T(\frac{n}{2^i}) + 1$$

$$T(n) = T(\tfrac{n}{2}) + 1$$
$$T(\tfrac{n}{2}) = T(\tfrac{n}{2^2}) + 1$$
$$T(\tfrac{n}{2^2}) = T(\tfrac{n}{2^3}) + 1$$
$$\ldots\ldots$$
$$\ldots\ldots$$
$$T(\tfrac{n}{2^{i-1}}) = T(\tfrac{n}{2^i}) + 1$$

Adding all these equations, we get

$$T(n) = T(\tfrac{n}{2^i}) + 1$$

$Let 2^i = n. i = \log_2 n$

$$T(n) = T(1) + \log_2 n.$$

If there is only one element in the array, it takes no time to search. Thus, $T(n) = \log_2 n$.

So, the complexity of the algorithm is $O(\log_2 n)$.

12.4.3 Linear Time Complexity

An algorithm is said to be linear if its running time increases linearly with the size of the input. Linear time complexity is denoted as $O(n)$.

Example 3.26 Find the time complexity of linear search.

Solution: Linear search is the simplest method of the searching technique. In this technique, the item to be searched is searched sequentially from the first item of the list until it is found. If the item is found, the location of the item in the list is returned. Following is the algorithm for linear search.

```
Procedure: Linear Search(List[ ], target)
    Inputs: List[ ] - A list of numbers
    Local: i integers
    Begin:
    int location = -1;
    For i = 0 to List.Size;
      If (List[i] == target)
      location = i;
      return location;
      End If
    Next i
    Return -1
End
```

Start calculating the complexity of the algorithm. The assignment of the value of 'i' to 'location' is of constant time, i.e., O(1). Returning the location also takes O(1). These two statements are executed once if the condition within the 'if' statement holds well. 'If' condition checking takes O(1). The statement 'if' takes maximum time if the condition is true [till the time is O(1) as O(1) + O(1) + O(1) = O(1)]. This 'if' statement is surrounded by a 'for' loop. The 'if' statement is executed as many times as the 'For' loop iterates. For the worst case (if the target element is the last element of list[ ] or if it does not exist), the loop iterates N times, where N is the size of list[ ].

Thus, the time taken by this algorithm is (worst case):

$$\sum_{i=0}^{N-1} \left(\underset{IF}{1} \right) = N$$

So, the complexity is O(N). Here N is a polynomial of N of degree 1.

12.4.4 Quasi linear Time Complexity

Before discussing quasilinear time complexity, we need the knowledge of linearithmic algorithm. The running time of the algorithms with linearithmic time complexity increases faster than the linear algorithm and slower than the logarithmic algorithm. Linearithmic time complexity is denoted as $O(n \log n)$.

Quasi linear time complexity was first proposed by Schnorr and Gurevich and Shelah in their research paper 'Satisfi ability is quasi linear complete in NQL' in 1978. An algorithm is said to be in quasilinear time complexity if its running time is of $O(n(logn)^k)$ where $k \geq 1$. Quasilinear algorithm runs faster than the polynomial algorithms of degree more than 1. Linearithmic algorithms are one type of quasilinear algorithm with k = 1.

Example 3.26 Find the time complexity of quicksort.

Solution: *Quicksort* is one type of divide and conquer type sorting algorithm proposed by T. Hoare. In this algorithm, a pivot element is chosen and the remaining elements of the list are divided into two halves based on the condition whether the elements are greater or smaller than the pivot element. The same quicksort algorithm is run again separately for two halves.

Process:

i) Take an element, called pivot element, from the list.

ii) Rearrange the list in such a way that all lesser elements than the pivot come before the pivot, while all greater elements than the pivot come after it (equal values can go in either direction). This partitioning makes the pivot to be placed in its fi nal position. This is called the **partition** operation.

iii) Recursively, sort the sub-list of lesser elements and the sub-list of greater elements.

A list of size zero or one never needs to be sorted. This is the base case of the recursion.

```
Procedure: QuickSort (A[ ], left, right)
    {
        if (left < right)
            select PIVOT s.t left ≤ PIVOT ≤ right
            PIVOT = PARTITION(A[ ], left, right, PIVOT)
            QuickSort(A[ ], left, PIVOT-1)
            QuickSort(A[ ], PIVOT + 1, right)
    }
```

```
Function: PARTITION(A[ ], left, right, PIVOT)
    {
        Value = A[PIVOT]
        swap A[PIVOT] and A[right]          / Move pivot to
end
        Index = left
        for i from left to right - 1 / left ≤ i < right
            if A[i] ¡ pivot
                swap A[i] and A[Index]
                Index = Index + 1
        swap A[Index] and A[right]          / Move pivot to its
final place return Index
}
```

12.4.5 Average Case Time Complexity

Let the pivot element be the kth smallest element of the array. Then, there are (k - 1) elements to the left of the position of k and (n - k) elements are at the right of the position of k. Thus, for sorting the left half of the array needs T(k - 1), and for sorting the right half of the array needs T(n - k). To place the pivot element, the number of comparisons is (n + 1). For an array of n element, the time complexity for average case quicksort is

$$T(n) = (n+1) + \frac{1}{n} \sum_{1 \le k \le n} T(k-1) + T(n-k) \tag{1}$$

$$nT(n) = n(n+1) + \sum_{1 \le k \le n} T(k-1) + T(n-k) \tag{2}$$

Replacing n by (n - 1), we get

$$(n-1)T(n-1) = n(n-1) + \sum_{1 \le k \le (n-1)} T(k-1) + T(n-k-1) \tag{3}$$

Subtracting equation (3) from (2), we get

$$nT(n)-(n-1)T(n-1)=2n+2T(n-1)$$

Note:

$$\sum_{1\le k\le n} T(k-1)+T(n-k)-\sum_{1\le k\le (n-1)} T(k-1)+T(n-k-1)$$

$$[T(0)+T(n-1)+T(1)+T(n-2)+.....+T(n-1)+$$
$$T(0)]-[T(0)+T(n-2)$$
$$=$$
$$+T(1)+T(n-3)+.....+T(n-2)+T(0)]$$
$$=2T(n-1)$$

Rearranging equation (4), we get

$$\Rightarrow nT(n) = 2n + (n+1)T(n-1)$$

$$\Rightarrow T(n) = 2 + \frac{(n+1)}{n}T(n-1)$$

$$\Rightarrow \frac{T(n)}{n+1} = \frac{1}{n}T(n-1) + \frac{2}{n+1}$$

$$= \frac{1}{n-2}T(n-3) + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{1}{n-2} T(n-3) + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{(n+1)}$$

$$\vdots$$

$$= \frac{T(1)}{2} + 2\left[\sum_{3 \le k \le n+1} \frac{1}{k}\right] \le \int_{2}^{n+1} \frac{1}{k}$$

$$\le [\log x]_2^{n+1} = \log (n+1) - \log_2 2 \le \log (n+1)$$

$$T(n) \le (n+1) \log (n+1)$$

Hence, the time complexity is $O(n \log n)$.