Searching

1. Linear Search

Suppose you have to search for a key *k* among a set of *n* distinct data items. These data items are stored in an array A[n]. We will first see a simple search strategy that compares each data item with the key *k* and reports success as soon as it finds the match. This search strategy is known as linear search. The main function is known as the driver function that invokes the linear search function. The code is as follows:

```
#include<stdio.h>
#define MAX 1000
int linear_search(int key, int size, int A[])
{
    int i =0;
    int found =0;

    while((i<size)&&(found==0))
    {
        if(A[i]==key) found =1;
        else i++;
    }

    if(found==1) return(i);
    else return(-1);
}
```

```
int main()
{
    char c;
    int key, i, n, index,  A[MAX];

    printf("Enter the number of array elements\n");
    scanf("%d", &n);

    printf("Enter the array elements\n");

    for(i=0; i<n; i++)
     scanf("%d", &A[i]);

    printf("Enter the key value\n");
    scanf("%d", &key);

    index = linear_search(key, n, A);

    if(index<0) printf("unsuccessful search\n");
    else printf("The key is present in index location
= %d\n", index);

    c=getchar();
    return(0);

}
```

Time Complexity Analysis:

(a) **Worst Case:** If n is the number of data items then to search for a key value we may have to look till the last element of the array even in case of successful search and we have to exhaust all the array elements in case of unsuccessful search. Thus the worst case time complexity is O(n). [Here O( ) denotes order notation]

(b) **Best Case:** The key will be the first array element and hence O(1).

(c) Average Case: The key can be the first element of the array with probability 1/n, the key can be the second array element with probability 1/n, …., the key can be the last array element with probability 1/n. Thus the expected number of comparisons for successful search:

$$E(comparisons) = \sum_{i=1}^{n} ip_i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

2. Binary Search

The overhead of linear search is quite high when we have repeated queries to search for key values. The idea that is used to improvise the search strategy can be easily visualized when we search for a word in a dictionary or a name in a telephone directory. The property of the dictionary or the telephone directory that enables fast search is that these are all sorted. So to search faster we have to incur an additional overhead of sorting to enable fast queries.

The idea of binary search is to compare the key value with the middle element of the array. If the key is equal to the middle element the search is successful and we return the middle index, otherwise if the key is smaller than the middle element then the all elements to the right of the middle element (including it) are pruned for further search, otherwise if the key is larger than the middle element then the all elements to the left of the middle element (including it) are

pruned for further search. This process is repeated till we exhaust all array elements. Again the main function is the driver function that invokes the binary search function. The code is as follows:

Note: In the code we have assumed that input array A[ ] of data items are sorted in ascending order.

```
#include<stdio.h>
#define MAX 1000

int binary_search(int key, int size, int A[])
{
  int low = 0;
  int high = size-1;
  int mid;
  int found =0;

  while((low<high)&&(found==0))
  {
      mid = (low+high)/2;
      if(A[mid]==key) found = 1;
      else if (A[mid] < key) low=mid+1;
            else high = mid-1;
  }

  if(found==1) return(mid);
  else return(-1);
}
```

```
int main()
{
  char c;
  int key, i, n, index,  A[MAX];

  printf("Enter   the   number   of   array
elements\n");
  scanf("%d", &n);

  printf("Enter the array elements\n");

  for(i=0; i<n; i++)
   scanf("%d", &A[i]);

  printf("Enter the key value\n");
  scanf("%d", &key);

  index = binary_search(key, n, A);

  if(index<0)printf("unsuccessful search\n");
  else  printf("The  key  is  present  in  index
location = %d\n", index);

  c=getchar();
  return(0);

}
```

Time Complexity Analysis:

1. **Best Case:** The key value can be the first middle element and hence O(1).

2. **Worst Case:** The maximum number of elements left after 1st iteration is $n/2$. The maximum number of elements left after 2nd iteration is $n/4 = n/2^2$. Thus the maximum number of elements left after k^{th} iteration is $n/2^k$. The process stops when $n/2^k = 1$ i.e., $k = \log n$. Thus the worst case time complexity of binary search is O(log n).

Here we should note that if we have to search for m keys in an array of n data items the time complexities will be as follows:

(a) Linear Search : O(mn)

(b) Binary Search: O($n\log n + m\log n$)

In case of binary search the first O($n\log n$) term is due to the overhead of sorting n data items.