Searching in Arrays

One of the basic operations to be performed on an array is searching. Searching an array means to find a particular element in the array. The search can be used to return the position of the element or check if it exists in the array.

Linear Search

The simplest search to be done on an array is the linear search. This search starts from one end of the array and keeps iterating until the element is found, or there are no more elements left (which means that the element does not exist).

There are no prerequisites for this search to work on an array. It can be used reliably in any situation.

Code for Linear Search

```
#include <stdio.h>

int main()
{
  int array[100], search, c, n;

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

   printf("Enter %d integer(s)\n", n);

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

   printf("Enter a number to search\n");
   scanf("%d", &search);

   for (c = 0; c < n; c++)
   {
     if (array[c] == search)
     {
       printf("%d is present at location %d.\n",
search, c+1);
       break;
     }
   }
   if (c == n)
     printf("%d isn't present in the array.\n", 
search);
   return 0;
}
</stdio.h>
```

Best use case

This search is best used when the list of elements is unsorted and the search is to be performed only once. It is also preferred for list to be small, as the time taken grows with the size of the data.

Time Complexity

- **Average**: O(n)
- **Best**: O(1)
- **Worst:** O(n)

The average time complexity is O(n) as the element to be found may be present at the end of the list or may not be present at all. Linear search has to visit all the elements until the needed element is found. Algorithmically, this comes out to be O(n).

The best and worst case of a search algorithm will be O(1) and O(n) respectively, as the element to be searched could always be found on the first iteration or the last iteration.

Binary Search

The linear search approach has one disadvantage. Finding elements in a large array will be time consuming. As the array grows, the time will increase linearly. A binary search can be used as a solution to this problem in some cases.

The principle of binary search is how we find a page in book. We open the book at a random page in the middle and based on that page we narrow our search to the left or right of the book. Indeed, this only is possible if the page numbers are in order.

Binary Search

Hence, the prerequisite of performing a binary search is that the array must be sorted. That is why this works only in cases where keeping a sorted copy of the array is possible.

The search starts by accessing the middle of the array. If the element is less than this element, it starts its search from this element to the left of the array. If the element is larger more than this element, it starts its search from this element to the right of the array. This process is repeated unless a the middle element is equal to the number we are searching.

Code for Binary Search

```
#include <stdio.h>

int main()
{
   int c, first, last, middle, n, search, array[100];

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

    printf("Enter all %d integers in sorted order\n", n);

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

    printf("Enter value to find\n");
    scanf("%d", &search);

    first = 0;
    last = n - 1;
    middle = (first+last)/2;

    while (first <= last) {
       if (array[middle] < search)
          first = middle + 1;
       else if (array[middle] == search) {
          printf("%d found at location %d.\n", search, middle+1);
          break;
       }
```

```
        else
            last = middle - 1;

        middle = (first + last)/2;
    }
    if (first > last)
        printf("%d is not found in the array.\n",
  search);

    return 0;
}
</stdio.h>
```

Best use case

This search is best used when the list of elements is already sorted (not always feasible, especially when new elements are frequently being inserted). The list to be searched can be very large without much decrease in searching time, due to the logarithmic time complexity of the algorithm.

Time Complexity

- **Average**: O(log n)
- **Best**: O(1)
- **Worst**: O(n)

The average time complexity of this algorithm of searching is O(log n) as the number of elements to search halves during each iteration.

The best and worst case of a search algorithm will be O(1) and O(n) respectively, as the element to be searched could always be found on the first iteration or the last iteration.