Linear Search

Linear Search Algorithm:

0. Start.

1. Ask the user to enter the size of the array.

2. Create an array of the specified size.

3. Ask the user to input elements into the array.

4. Ask the user to enter the element to search for.

5. Initialize a flag variable to track if the element is found.

6. Iterate through each element of the array:

   - Check if the current element is equal to the search key.

   - If it is, set the flag to 1 and break out of the loop.

7. After the loop, check the flag value:

   - If the flag is 1, print "Element found in the array."

   - If not, print "Element not found in the array."

8. End.

Linear Search Program:

```
#include <stdio.h>

int main()
{
    int n, key;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int ar[n];
    printf("\nEnter the elements of the array: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ar[i]);
    }
```

```c
    printf("\nEnter the element to search: ");
    scanf("%d", &key);
    int flag = 0;
    for (int i = 0; i < n; i++)
    {
        if (ar[i] == key)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
    {
        printf("Element found in the array.\n");
    }
    else
    {
        printf("Element not found in the array.\n");
    }
    return 0;
}
```

Linear Search Complexity:

Worst-case time complexity: O(n) where n is the size of the array.

Space complexity: O(n) where n is the size of the array.

Binary Search

Binary Search Algorithm:

0. Start.

1. Ask the user to enter the size of the array.

2. Create an array of the specified size.

3. Ask the user to input elements into the array.

4. Ask the user to enter the element to search for.

5. Initialize variables for the low index, high index, and a flag to track if the element is found.

6. Start a loop that continues until the low index is less than or equal to the high index.

7. Calculate the middle index of the current range.

8. Check if the middle element is equal to the element being searched for:

   - If yes, set the flag to 1 and break out of the loop.

   - If not, adjust the range based on whether the middle element is less than or greater than the search element.

9. After exiting the loop, check if the flag is set:

   - If yes, print "Element found."

   - If not, print "Element not found."

10. End.

Binary Search Program:

```
#include <stdio.h>
int main(){
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int ar[n];
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ar[i]);
```

```c
    }
    int key;
    printf("Enter the element to search: ");
    scanf("%d", &key);
    int low = 0, high = n - 1, flag = 0;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (ar[mid] == key)
        {
            flag = 1;
            break;
        }
        else if (ar[mid] < key)
        {
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }
    if (flag == 1)
    {
        printf("Element found.\n");
    }
    else
    {
        printf("Element not found.\n");
    }
    return 0;
}
```

Binary Search Complexity:

During the first iteration, the element is searched in the entire array. Therefore, length of the array = n.

In the second iteration, only half of the original array is searched. Hence, length of the array = n/2.

In the third iteration, half of the previous sub-array is searched. Here, length of the array will be = n/4.

Similarly, in the ith iteration, the length of the array will become n/2i

To achieve a successful search, after the last iteration the length of array must be 1. Hence,

n/2i = 1

That gives us –> n = 2i

Applying log on both sides,

log n = log 2i

log n = i log 2

i = log n

Worst-case time complexity: O(log n)

Space complexity: O(n)

Bubble Sort

Bubble Sort Algorithm:

0. Start.

1. Ask the user to enter the size of the array.

2. Create an array of the specified size.

3. Ask the user to input elements into the array.

4. Perform a nested loop to compare elements for sorting:

   - Loop through each element in the array except the last one.

   - For each element, compare it with all the elements after it.

   - If an element is greater than the element being compared with, swap them.

5. Print "Sorted array:".

6. Print the sorted array elements.

7. End.

Bubble Sort Program:

```
#include <stdio.h>
int main()
{
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int ar[n];
    printf("\nEnter the elements of the array: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ar[i]);
    }
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (ar[i] > ar[j])
            {
                int temp = ar[i];
                ar[i] = ar[j];
                ar[j] = temp;
            }
        }
    }
    printf("\nSorted array: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ar[i]);
    }
    return 0;
}
```

Bubble Sort Complexity:

1 + 2 + 3 + ... + (n - 1) = n(n - 1)/2 = O(n^2)

Worst-case time complexity: O(n^2)

Space complexity: O(n)

Insertion Sort

Insertion Sort Algorithm:

0. Start.

1. Ask the user to enter the size of the array.

2. Create an array of the specified size.

3. Ask the user to input elements into the array.

4. Perform insertion sort on the array:

   - Start a loop from the second element to the last element of the array.

   - Store the current element in a variable key.

   - Initialize a variable j to the index before the current element.

   - While j is greater than or equal to 0 and the element at index j is greater than the key:

     - Shift elements to the right to make space for the key.

     - Decrement j.

   - Insert the key into its correct sorted position.

5. Print "Sorted array:".

6. Print the sorted array elements.

7. End.

Insertion Sort Program:

```
#include <stdio.h>
int main()
{
    int n;
    printf("Enter the size: ");
    scanf("%d", &n);
```

```
    int ar[n];
    printf("\nEnter the array elements: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ar[i]);
    }
    for (int i = 1; i < n; i++)
    {
        int key = ar[i];
        int j = i - 1;
        while (j >= 0 && ar[j] > key)
        {
            ar[j + 1] = ar[j];
            j = j - 1;
        }
        ar[j + 1] = key;
    }
    printf("\nSorted array: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ar[i]);
    }
    return 0;
}
```

Insertion Sort Complexity:

when the array is sorted in reverse order, the Insertion Sort algorithm will take its maximum number of comparisons and swaps. This results in a time complexity of O(n^2), where n is the number of elements in the array.

(n-1)+ (n-2) +...+3+2+1= (n-1)* n/2 = (n*2n-2)/2 =(2n^2-2)/2

Worst-case time complexity: O(n^2)

Space complexity: O(n).

Merge Sort

Merge Sort Algorithm:

0. Start.

1. Ask the user to enter the size of the array.

2. Create an array of the specified size.

3. Ask the user to input elements into the array.

4. Define a function `Merge` to merge two sorted subarrays.

   - Calculate the sizes of the two subarrays.

   - Create temporary arrays `Left` and `Right`.

   - Copy elements from the main array to the temporary arrays.

   - Merge the two subarrays back into the main array in sorted order.

5. Define a function `Merge_Sort` for performing merge sort.

   - If left index is less than the right index:

     - Calculate the middle index.

     - Recursively call `Merge_Sort` on the two halves of the array.

     - Merge the two sorted halves using the `Merge` function.

6. In the `main` function:

   - Call `Merge_Sort` function on the array to sort it.

   - Print "The sorted array is:".

   - Print the sorted array elements.

7. End.

Merge Sort Program:

```
#include <stdio.h>
void Merge(int arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int Left[n1], Right[n2];
    for (int i = 0; i < n1; i++)
```

```
{
    Left[i] = arr[left + i];
}
for (int j = 0; j < n2; j++)
{
    Right[j] = arr[mid + 1 + j];
}
int i = 0, j = 0, k = left;
while (i < n1 && j < n2)
{
    if (Left[i] <= Right[j])
    {
        arr[k] = Left[i];
        i++;
    }
    else
    {
        arr[k] = Right[j];
        j++;
    }
    k++;
}
while (i < n1)
{
    arr[k] = Left[i];
    i++;
    k++;
}
while (j < n2)
{
    arr[k] = Right[j];
    j++;
```

```
        k++;
    }
}
void Merge_Sort(int *ar, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        Merge_Sort(ar, left, mid);
        Merge_Sort(ar, mid + 1, right);
        Merge(ar, left, mid, right);
    }
}

int main()
{
    int n;
    printf("Enter the size: ");
    scanf("%d", &n);
    int ar[n];
    printf("\nEnter the elements of array: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ar[i]);
    }
    Merge_Sort(ar, 0, n - 1);
    printf("\nThe sorted array is: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ar[i]);
    }
    return 0;
```

```
}
```

Merge Sort Complexity:

Worst-case time complexity: O(n log n)

Space complexity: O(n)