Linear search:

Program 1:

#include <stdio.h>

int search(int arr[], int N, int x)

{

    int i;

    for (i = 0; i < N; i++)

        if (arr[i] == x)

            return i;

    return -1;

}

// Driver code

int main(void)

{

    int arr[] = { 2, 3, 4, 10, 40 };

    int x = 10;

    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call

    int result = search(arr, N, x);

    (result == -1)

        ? printf("Element is not present in array")

        : printf("Element is present at index %d", result);

    return 0;

}

Program 2:

#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)    */\* If required element is found \*/*  
    {  
      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;  
}

**Complexity Analysis of Linear Search:**

**Time Complexity:**

* **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
* **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst case complexity is O(N) where N is the size of the list.
* **Average Case:** O(N)

Binary Search:

Program 1:With using function

// Binary Search in C

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {

// Repeat until the pointers low and high meet each other

while (low <= high) {

int mid = low + (high - low) / 2;

if (array[mid] == x)

return mid;

if (array[mid] < x)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main(void) {

int array[] = {3, 4, 5, 6, 7, 8, 9};

int n = sizeof(array) / sizeof(array[0]);

int x = 4;

int result = binarySearch(array, x, 0, n - 1);

if (result == -1)

printf("Not found");

else

printf("Element is found at index %d", result);

return 0;

}

Program 2:

#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 %d integers**\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("Not found! %d isn't present in the list.**\n**", search);

  return 0;  
}

## Binary Search Complexity Analysis:

The time and space complexities of the binary search algorithm are mentioned below.

### ****Time Complexity of Binary Search Algorithm:****

* **Best Case:** O(1)

*Best case is when the element is at the middle index of the array. It takes only one comparison to find the target element. So the best case complexity is****O(1)****.*

* **Average Case:** O(log N)

*Consider array****arr[]****of length****N****and element****X****to be found. There can be two cases:*

* *Case1: Element is present in the array*
* *Case2: Element is not present in the array.*

*There are****N****Case1 and****1****Case2. So total number of cases =****N+1****. Now notice the following:*

* *An element at index N/2 can be found in****1****comparison*
* *Elements at index N/4 and 3N/4 can be found in****2****comparisons.*
* *Elements at indices N/8, 3N/8, 5N/8 and 7N/8 can be found in****3****comparisons and so on.*

*Based on this we can conclude that elements that require:*

* *1 comparison = 1*
* *2 comparisons = 2*
* *3 comparisons = 4*
* ***x****comparisons =****2x-1****where****x****belongs to the range****[1, logN]****because maximum comparisons = maximum time N can be halved = maximum comparisons to reach 1st element = logN.*

*So, total comparisons  
= 1\*(elements requiring 1 comparisons) + 2\*(elements requiring 2 comparisons) + . . . + logN\*(elements requiring logN comparisons)  
= 1\*1 + 2\*2 + 3\*4 + . . . + logN \* (2logN-1)  
= 2logN \* (logN – 1) + 1  
=****N \* (logN – 1) + 1***

*Total number of cases =****N+1****.*

*Therefore, the average complexity = (****N\*(logN – 1) + 1)/N+1****=****N\*logN / (N+1) + 1/(N+1)****. Here the dominant term is N\*logN/(N+1) which is approximately****logN****. So the average case complexity is****O(logN)***

* **Worst Case:** O(log N)

**Bubble Sort**:

Program 1:

#include <stdio.h>

int main(){

    int arr[50], num, x, y, temp;

    printf("Please Enter the Number of Elements you want in the array: ");

    scanf("%d", &num);

    printf("Please Enter the Value of Elements: ");

    for(x = 0; x < num; x++)

        scanf("%d", &arr[x]);

    for(x = 0; x < num - 1; x++){

        for(y = 0; y < num - x - 1; y++){

            if(arr[y] > arr[y + 1]){

                temp = arr[y];

                arr[y] = arr[y + 1];

                arr[y + 1] = temp;

            }

        }

    }

    printf("Array after implementing bubble sort: ");

    for(x = 0; x < num; x++){

        printf("%d  ", arr[x]);

    }

    return 0;

}

Program 2:

// C program for implementation of Bubble sort

#include <stdio.h>

void swap(int\* xp, int\* yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

    int i, j;

    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place

        for (j = 0; j < n - i - 1; j++)

            if (arr[j] > arr[j + 1])

                swap(&arr[j], &arr[j + 1]);

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

// Driver program to test above functions

int main()

{

    int arr[] = { 5, 1, 4, 2, 8 };

    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

Selection Sort:

Let's take an array {20, 12 , 23, 55 ,21}

* Set the first element of the array as minimum.

Minimum = 20

* Compare the minimum with the next element, if it is smaller than minimum assign this element as minimum. Do this till the end of the array.

Comparing with 12 : 20 > 12 , minimum = 12

Comparing with 23 : 12 < 23 , minimum = 12

Comparing with 55 : 12 < 55 , minimum = 12

Comparing with 21 : 12 < 21 , minimum = 12

* Place the minimum at the first position( index 0) of the array.

Array = {12, 20 ,23, 55, 21}

* for the next iteration, start sorting from the first unsorted element i.e. the element next to where the minimum is placed.

Array = {12, 20 ,23, 55, 21}

Searching starts from 20, next element where minimum is placed.

Iteration 2 :

Minimum = 20

Comparing with 23 : 20 < 23 , minimum = 20

Comparing with 55 : 20 < 55 , minimum = 20

Comparing with 21 : 20 < 21 , minimum = 20

Minimum in place no change,

Array = {12, 20 ,23, 55, 21}

Iteration 3 :

Minimum = 23.

Comparing with 55 : 23 < 55 , minimum = 23

Comparing with 21 : 23 > 21 , minimum = 21

Minimum is moved to index = 2

Array = {12, 20, 21, 55, 23}

Iteration 4 :

Minimum = 55

Comparing with 23 : 23 < 55 , minimum = 23

Minimum in moved to index 3 Array = {12, 20, 21, 23, 55}

**Complexity Analysis of Selection Sort:**

**Time Complexity:** The time complexity of Selection Sort is O(N2) as there are two nested loops:

* One loop to select an element of Array one by one = O(N)
* Another loop to compare that element with every other Array element = O(N)

Therefore overall complexity = O(N) \* O(N) = O(N\*N) = O(N2)

Program 1:

#include <stdio.h>

int main() {

   int arr[10]={6,12,0,18,11,99,55,45,34,2};

   int n=10;

   int i, j, position, swap;

   for (i = 0; i < (n - 1); i++) {

      position = i;

      for (j = i + 1; j < n; j++) {

         if (arr[position] > arr[j])

            position = j;

      }

      if (position != i) {

         swap = arr[i];

         arr[i] = arr[position];

         arr[position] = swap;

      }

   }

   for (i = 0; i < n; i++)

      printf("%d\t", arr[i]);

   return 0;

}

Program 2:

// C program for implementation of selection sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    // One by one move boundary of unsorted subarray

    for (i = 0; i < n-1; i++)

    {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i+1; j < n; j++)

          if (arr[j] < arr[min\_idx])

            min\_idx = j;

        // Swap the found minimum element with the first element

           if(min\_idx != i)

            swap(&arr[min\_idx], &arr[i]);

    }

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

// Driver program to test above functions

int main()

{

    int arr[] = {64, 25, 12, 22, 11};

    int n = sizeof(arr)/sizeof(arr[0]);

    selectionSort(arr, n);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

Insertion sort:

## Algorithm for Insertion Sort

* **Step 1** − If the element is the first one, it is already sorted.
* **Step 2** – Move to next element
* **Step 3** − Compare the current element with all elements in the sorted array
* **Step 4** – If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right
* **Step 5** − Insert the value at the correct position
* **Step 6** − Repeat until the complete list is sorted

Let’s understand how insertion sort is working in the above image.

* **122**, 17, 93, 3, 36

for i = 1(2nd element) to 36 (last element)

i = 1. Since 17 is smaller than 122, move 122 and insert 17 before 122

* **17, 122**, 93, 3, 36

i = 2. Since 93 is smaller than 122, move 122 and insert 93 before 122

* **17, 93,122**, 3, 36

i = 3. 3 will move to the beginning and all other elements from 17 to 122 will move one position ahead of their current position.

* **3, 17, 93, 122,** 36

i = 4. 36 will move to position after 17, and elements from 93 to 122 will move one position ahead of their current position.

* **3, 17, 36, 93 ,122**

Now that we have understood how Insertion Sort works, let us quickly look at a C code to implement insertion sort

Program 1:

#include <stdio.h>

int main()  
{  
  int n, array[1000], c, d, min, flag = 0;

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

  printf("Enter %d integers**\n**", n);

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

  for (c = 1 ; c <= n - 1; c++) {  
    min = array[c];

    for (d = c - 1 ; d >= 0; d--) {  
      if (array[d] > min) {  
        array[d+1] = array[d];  
        flag = 1;  
      }  
      else  
        **break**;  
    }  
    if (flag)  
      array[d+1] = min;  
  }

  printf("Sorted list in ascending order:**\n**");

  for (c = 0; c <= n - 1; c++) {  
    printf("%d**\n**", array[c]);  
  }

  return 0;  
}

Program 2:

// C program for insertion sort

#include <math.h>

#include <stdio.h>

/\* Function to sort an array

   using insertion sort\*/

void insertionSort(int arr[], int n)

{

    int i, min, j;

    for (i = 1; i < n; i++)

    {

        min = arr[i];

        j = i - 1;

        while (j >= 0 && arr[j] > min)

        {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = min;

    }

}

// A utility function to print

// an array of size n

void printArray(int arr[], int n)

{

    int i;

    for (i = 0; i < n; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

// Driver code

int main()

{

    int arr[] = {12, 11, 13, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);

    printArray(arr, n);

    return 0;

}

Best case complexity of insertion sort is O(n):If the intial data is sorted ,only one compareison is made in each pass,

Total number of comparisions:n-1

average and the worst case complexity is O(n2):

Total number of comparisions=1+2+..+n-1=n(n-1)/2

Complexity: O(n2)

Merge Sort:

**Merge Sort** is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides the input array into two halves, calls itself for the two halves, and then it merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

**Algorithm:**

**Step 1:** Start

**Step 2:** Declare an array and left, right, mid variable

**Step 3:** Perform merge function.

mergesort(array,left,right)

mergesort (array, left, right)

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

**Step 4:** Stop

Program 1:

#include <stdio.h>

void printArray(int \*A, int n)

{

for (int i = 0; i < n; i++)

{

printf("%d ", A[i]);

}

printf("\n");

}

void merge(int A[], int mid, int low, int high)

{

int i, j, k, B[100];

i = low;

j = mid + 1;

k = low;

while (i <= mid && j <= high)

{

if (A[i] < A[j])

{

B[k] = A[i];

i++;

k++;

}

else

{

B[k] = A[j];

j++;

k++;

}

}

while (i <= mid)

{

B[k] = A[i];

k++;

i++;

}

while (j <= high)

{

B[k] = A[j];

k++;

j++;

}

for (int i = low; i <= high; i++)

{

A[i] = B[i];

}

}

void mergeSort(int A[], int low, int high){

int mid;

if(low<high){

mid = (low + high) /2;

mergeSort(A, low, mid);

mergeSort(A, mid+1, high);

merge(A, mid, low, high);

}

}

int main()

{

// int A[] = {9, 14, 4, 8, 7, 5, 6};

int A[] = {9, 1, 4, 14, 4, 15, 6};

int n = 7;

printArray(A, n);

mergeSort(A, 0, 6);

printArray(A, n);

return 0;

}

### Time Complexity

Best Case Complexity: O(n\*log n)

Worst Case Complexity: O(n\*log n)

Average Case Complexity: O(n\*log n)

Quick Sort:

Quicksort is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) based on the **divide and conquer approach** where

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).  
     
   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.
4. #include<stdio.h>
5. void quicksort(int number[25],int first,int last){
6. int i, j, pivot, temp;
7. if(first<last){
8. pivot=first;
9. i=first;
10. j=last;
11. while(i<j){
12. while(number[i]<=number[pivot]&&i<last)
13. i++;
14. while(number[j]>number[pivot])
15. j--;
16. if(i<j){
17. //swapping of elements at i and j
18. temp=number[i];
19. number[i]=number[j];
20. number[j]=temp;
21. }
22. }
23. //swapping of element at pivot and j
24. temp=number[pivot];
25. number[pivot]=number[j];
26. number[j]=temp;
27. quicksort(number,first,j-1);
28. quicksort(number,j+1,last);
29. }
30. }
31. int main(){
32. int i, count, number[25];
33. printf("How many elements are u going to enter?: ");
34. scanf("%d",&count);
35. printf("Enter %d elements: ", count);
36. for(i=0;i<count;i++)
37. scanf("%d",&number[i]);
38. quicksort(number,0,count-1);
39. printf("Order of Sorted elements: ");
40. for(i=0;i<count;i++)
41. printf(" %d",number[i]);
42. return 0;
43. }