# Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. In such a way that

put the pivot at its correct position in sorted array and put all smaller elements (smaller than pivot) before pivot, and put all greater elements (greater than pivot) after pivot. All this should be done in linear time.

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

Pseudo Code for recursive QuickSort function :

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

// find the correct position of the pivot element

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

**Partition Algorithm**  
we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

Pseudo code for partition()

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

i = (low - 1) // Index of smaller element

for (j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

}

}

swap arr[i + 1] and arr[high])

return (i + 1)

}

**private** **int**[] quickSort(**int**[] input, **int** start, **int** end) {

// if start is less than end than only input has to sort.Otherwise sorting of the list is done.

**if** (start < end) {

// get the correct index of the pivot element

**int** pivotExactIndex = partition(input, start, end);

// sort the left hand side list

quickSort(input, start, pivotExactIndex - 1);

// sort the right hand side list

quickSort(input, pivotExactIndex + 1, end);

}

//return the sorted list

**return** input;

}

**private** **int** partition(**int**[] input, **int** start, **int** end) {

// taking last element as the pivot

**int** pivot = input[end];

// index which will be used for swapping with smaller element

**int** i = start - 1;

// iterating the list

**for** (**int** j = start; j < end; j++) {

// if the current index is less than and equal to pivot

**if** (input[j] <= pivot) {

// incrementing the swapping index

// because till this all element is less than the pivot

// so incrementing it and then swapping with next small element

// till here swapping index all element will less than pivot

++i;

// swap the small element

swap(input, i, j);

}

}

// when iteration reach to (pivot-1) then swapping the pivot with (swapping index + 1)

// pivot will be at the swapping index and before that all element will be less than pivot

// and after that all element will be greater than pivot

swap(input, ++i, end);

// return the pivot index

**return** i;

}

**private** **void** swap(**int**[] input, **int** i, **int** j) {

**int** temp = input[i];

input[i] = input[j];

input[j] = temp;

}

# Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the list into smallest units, by dividing the list into two halves at one time.

Then start merging all smallest unit to get the sorted list by merging two list on comparing its element.

Merge Sort Algorithm:

/\* l is for left index and r is right index of the

   sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

    if (l < r)

    {

        // Same as (l+r)/2, but avoids overflow for

        // large l and h

        int m = l+(r-l)/2;

        // Sort first and second halves

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

Merge Algorithm:

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    /\* create temp arrays \*/

    int L[n1], R[n2];

    /\* Copy data to temp arrays L[] and R[] \*/

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /\* Merge the temp arrays back into arr[l..r]\*/

    i = 0; // Initial index of first subarray

    j = 0; // Initial index of second subarray

    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    /\* Copy the remaining elements of L[], if there

       are any \*/

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of R[], if there

       are any \*/

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

**private** **void** mergeSort(**int**[] input, **int** start, **int** end) {

**if** (start < end) {

**int** m = (start + end) / 2;

mergeSort(input, start, m);

mergeSort(input, m + 1, end);

merge(input, start, m, end);

}

}

**private** **void** merge(**int**[] input, **int** start, **int** m, **int** end) {

// find size of the two array

**int** ln = m - start + 1;// since middle point(m) is part of the left array that's why length is

// m-start+1

**int** rn = end - m;// length is end-m because it will start from m+1

// create two array

**int**[] lArr = **new** **int**[ln];

**int**[] rArr = **new** **int**[rn];

// copy elements to left array

**for** (**int** i = 0; i < ln; i++) {

lArr[i] = input[start + i];

}

// copy elements to right array

**for** (**int** j = 0; j < rn; j++) {

rArr[j] = input[m+1+j];

}

// index for iterating left and right array

**int** i = 0, j = 0;

//index of merged subarray

**int** k = start;

// compare both array and put the smaller in the input array

**while**(i < ln && j < rn) {

**if**(lArr[i] <= rArr[j]) {

input[k] = lArr[i];

i++;

} **else** {

input[k] = rArr[j];

j++;

}

k++;

}

// if left array has element left to put in input array

**while**(i<ln) {

input[k] = lArr[i];

i++;

k++;

}

// if right array has elemetn left to put in input array

**while**(j<rn) {

input[k] = rArr[j];

j++;

k++;

}

# }

# Insertion Sort

A sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.

**private** **int**[] insertionSort(**int**[] input) {

**int** n = input.length;

**for**(**int** j=1;j<n;j++) {

**int** key = input[j];

**int** i=j-1;

**while**(i>-1 && input[i]>key) {

input[i+1] = input[i];

i--;

}

input[i+1]=key;

}

**return** input;

}

# Bubble Sort

It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.This algorithm's average time and worst-case performance is O(n2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty).

**private** **int**[] bubbleSort(**int**[] input) {

**for**(**int** i = 0; i< input.length ; i++) {

**for**(**int** j = 1,k=0; j<input.length-i;j++,k++) {

**if**(input[k]>input[j]) {

swap(input, k, j);

}

}

}

**return** input;

}

**private** **void** swap(**int**[] input, **int** i, **int** j) {

**int** temp = input[j];

input[j]= input[i];

input[i] = temp;

}

void bubbleSort(int arr[])

    {

        int n = arr.length;

        for (int i = 0; i < n-1; i++)

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

                if (arr[j] > arr[j+1])

                {

                    // swap temp and arr[i]

                    int temp = arr[j];

                    arr[j] = arr[j+1];

                    arr[j+1] = temp;

                }

    }

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Time** | | |  | | | |
| Sort | Average | Best | Worst | Space | Stability | Remarks |  |
| [Bubble sort](https://www.cprogramming.com/tutorial/computersciencetheory/sorting1.html) | O(n^2) | O(n^2) | O(n^2) | Constant | Stable | Always use a modified bubble sort |  |
| [Modified Bubble sort](https://www.cprogramming.com/tutorial/computersciencetheory/sorting1.html) | O(n^2) | O(n) | O(n^2) | Constant | Stable | Stops after reaching a sorted array |  |
| [Selection Sort](https://www.cprogramming.com/tutorial/computersciencetheory/sorting2.html) | O(n^2) | O(n^2) | O(n^2) | Constant | Stable | Even a perfectly sorted input requires scanning the entire array |  |
| [Insertion Sort](https://www.cprogramming.com/tutorial/computersciencetheory/sorting2.html) | O(n^2) | O(n) | O(n^2) | Constant | Stable | In the best case (already sorted), every insert requires constant time |  |
| [Heap Sort](https://www.cprogramming.com/tutorial/computersciencetheory/heapsort.html) | O(n\*log(n)) | O(n\*log(n)) | O(n\*log(n)) | Constant | Instable | By using input array as storage for the heap, it is possible to achieve constant space |  |
| [Merge Sort](https://www.cprogramming.com/tutorial/computersciencetheory/mergesort.html) | O(n\*log(n)) | O(n\*log(n)) | O(n\*log(n)) | Depends | Stable | On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space |  |
| [Quicksort](https://www.cprogramming.com/tutorial/computersciencetheory/quicksort.html) | O(n\*log(n)) | O(n\*log(n)) | O(n^2) | Constant | Stable | Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array. |  |

|  |  |
| --- | --- |
|  |  |

# Searching

## What is Searching?

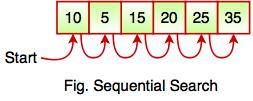
* Searching is the process of finding a given value position in a list of values.
* It decides whether a search key is present in the data or not.
* It is the algorithmic process of finding a particular item in a collection of items.
* It can be done on internal data structure or on external data structure.

## Searching Techniques

**To search an element in a given array, it can be done in following ways:**  
  
1. Sequential Search  
2. Binary Search

#### 1. Sequential Search

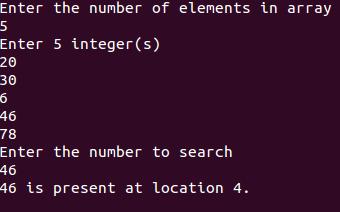
* Sequential search is also called as Linear Search.
* Sequential search starts at the beginning of the list and checks every element of the list.
* It is a basic and simple search algorithm.
* Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

  
  
The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.  
  
**The following code snippet shows the sequential search operation:**

function searchValue(value, target)  
{  
      for (var i = 0; i < value.length; i++)  
      {  
             if (value[i] == target)  
             {  
                     return i;  
             }  
      }  
      return -1;  
}  
searchValue([10, 5, 15, 20, 25, 35] , 25);   // Call the function with array and number to be searched

#### Example: Program for Sequential Search

#include <stdio.h>   
int main()   
{   
   int arr[50], search, cnt, num;   
  
   printf("Enter the number of elements in array\n");   
   scanf("%d",&num);   
  
   printf("Enter %d integer(s)\n", num);   
  
   for (cnt = 0; cnt < num; cnt++)   
   scanf("%d", &arr[cnt]);   
  
   printf("Enter the number to search\n");   
   scanf("%d", &search);   
  
   for (cnt = 0; cnt < num; cnt++)   
   {   
      if (arr[cnt] == search)     /\* if required element found \*/   
      {   
         printf("%d is present at location %d.\n", search, cnt+1);   
         break;   
      }   
   }   
   if (cnt == num)   
      printf("%d is not present in array.\n", search);   
  
   return 0;   
}

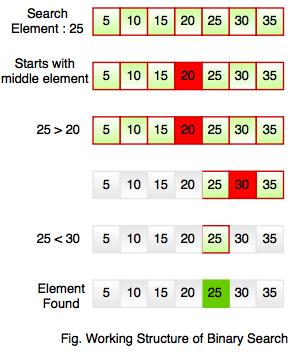
**Output:**  
  


#### 2. Binary Search

* Binary Search is used for searching an element in a sorted array.
* It is a fast search algorithm with run-time complexity of O(log n).
* Binary search works on the principle of divide and conquer.
* This searching technique looks for a particular element by comparing the middle most element of the collection.
* It is useful when there are large number of elements in an array.

binary array

* The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

**For example,** if searching an element 25 in the 7-element array, following figure shows how binary search works:  
  
  
  
Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

#### Example: Program for Binary Search

#include<stdio.h>   
#include<conio.h>   
  
void main()   
{   
   int f, l, m, size, i, sElement, list[50]; //int f, l ,m : First, Last, Middle  
   clrscr();   
  
   printf("Enter the size of the list: ");   
   scanf("%d",&size);   
  
   printf("Enter %d integer values : \n", size);   
  
   for (i = 0; i < size; i++)   
      scanf("%d",&list[i]);   
  
   printf("Enter value to be search: ");   
   scanf("%d", &sElement);   
  
   f = 0;   
   l = size - 1;   
   m = (f+l)/2;   
  
   while (f <= l) {   
      if (list[m] < sElement)   
         f = m + 1;      
      else if (list[m] == sElement) {   
         printf("Element found at index %d.\n",m);   
         break;   
      }   
      else   
         l = m - 1;    
      m = (f + l)/2;   
   }   
   if (f > l)   
      printf("Element Not found in the list.");   
   getch();    
}

**Output:**  
  
