**Linear Search Algorithm**

**What is Linear Search**

*Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set. It is the easiest searching algorithm*

**

Given an array arr[] of N elements, the task is to write a function to search a given element x in arr[].

**Examples:**

***Input:****arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}, x = 110;****Output:****6****Explanation:****Element x is present at index 6*

***Input:****arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}, x = 175;****Output:****-1****Explanation:****Element x is not present in arr[].*

Follow the below idea to solve the problem:

*Iterate from 0 to N-1**and compare the value of every index with x if they match return index*

 Follow the given steps to solve the problem:

* Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
* If x matches with an element, return the index.
* If x doesn’t match with any of the elements, return -1.

Below is the implementation of the above approach:

**Python3**

|  |
| --- |
| # Python3 code to linearly search x in arr[].  # If x is present then return its location,  # otherwise return -1  def search(arr, N, x):        for i in range(0, N):          if (arr[i] == x):              return i      return -1  # Driver Code  if \_\_name\_\_ == "\_\_main\_\_":      arr = [2, 3, 4, 10, 40]      x = 10      N = len(arr)      # Function call      result = search(arr, N, x)      if(result == -1):          print("Element is not present in array")      else:          print("Element is present at index", result) |

**Python3**

|  |
| --- |
| """Python Program to Implement Linear Search Recursively"""      def linear\_search(arr, key, size):          # If the array is empty we will return -1        if size == 0:          return -1        # Otherwise if the array consists of only one element and that element is not the one      # we are searching for then it will also return  -1        elif size == 1 and arr[0] != key:          return -1        # ELse , if the element at the size index is same as the element we are searching for      # Then return the size. This will return the index position is 0 index manner.      # i.e if the element is present at 6th position it will return 5.      # To get the exact position in human readble format (counting starts from 1 not 0)      # Then just return size + 1        elif arr[size] == key:          return size        # If none of the conditions are True then in else condition we will call the      # function recursively by decreasing the size by 1 each time.        else:          return linear\_search(arr, key, size-1)    # Driver's code  if \_\_name\_\_ == "\_\_main\_\_":      arr = [5, 15, 6, 9, 4]      key = 4      size = len(arr)-1        # Calling the Function      print("The element ", key, " is found at index: ",          linear\_search(arr, key, size), " of given array")      # Code Contributed By - DwaipayanBandyopadhyay |

**Binary Search**

**Problem:**Given a sorted array **arr[]** of **n** elements, write a function to search a given element **x** in**arr[]**and return the index of x in the array.

                 Consider array is 0 base index.

**Examples:**

***Input:****arr[] = {10, 20, 30, 50, 60, 80, 110, 130, 140, 170}, x = 110****Output:****6****Explanation:****Element x is present at index 6.*

***Input:****arr[] = {10, 20, 30, 40, 60, 110, 120, 130, 170}, x = 175****Output:****-1****Explanation:****Element x is not present in arr[].*

**Linear Search Approach**: A simple approach is to do a [**linear search**](https://www.geeksforgeeks.org/linear-search/)**.** The time complexity of the Linear search is O(n). Another approach to perform the same task is using *Binary Search*.

**Binary Search Approach:**

***Binary Search****is a*[*searching algorithm*](https://www.geeksforgeeks.org/searching-algorithms/)*used in a sorted array by****repeatedly dividing the search interval in half****. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).*

**Binary Search Algorithm:** The basic steps to perform Binary Search are:

* Begin with the mid element of the whole array as a search key.
* If the value of the search key is equal to the item then return an index of the search key.
* Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
* Otherwise, narrow it to the upper half.
* Repeatedly check from the second point until the value is found or the interval is empty.

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

1. Iteration Method

binarySearch(arr, x, low, high)

repeat till low = high

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > arr[mid]) // x is on the right side

low = mid + 1

else // x is on the left side

high = mid - 1

2. Recursive Method (The recursive method follows the divide and conquers approach)

binarySearch(arr, x, low, high)

if low > high

return False

else

mid = (low + high) / 2

if x == arr[mid]

return mid

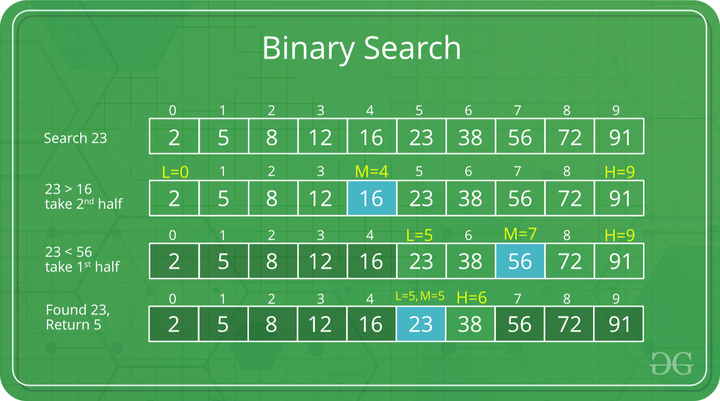
else if x > arr[mid] // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the right side

return binarySearch(arr, x, low, mid - 1)

**Illustration of Binary Search Algorithm:**



*Example of Binary Search Algorithm*

Recommended Problem

Searching an element in a sorted array

[Searching](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Searching&sortBy=submissions)

[Binary Search](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Binary%20Search&sortBy=submissions)

[Paytm](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Paytm&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/who-will-win-1587115621/1" \o "Permalink to Searching an element in a sorted array)

Submission count: 70.3K

**Step-by-step Binary Search Algorithm:** We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

**Recursive implementation of Binary Search**:

|  |
| --- |
| # Python3 Program for recursive binary search.    # Returns index of x in arr if present, else -1      def binarySearch(arr, l, r, x):        # Check base case      if r >= l:            mid = l + (r - l) // 2            # If element is present at the middle itself          if arr[mid] == x:              return mid            # If element is smaller than mid, then it          # can only be present in left subarray          elif arr[mid] > x:              return binarySearch(arr, l, mid-1, x)            # Else the element can only be present          # in right subarray          else:              return binarySearch(arr, mid + 1, r, x)        else:          # Element is not present in the array          return -1      # Driver Code  arr = [2, 3, 4, 10, 40]  x = 10    # Function call  result = binarySearch(arr, 0, len(arr)-1, x)    if result != -1:      print("Element is present at index % d" % result)  else:      print("Element is not present in array") |

**Output**

Element is present at index 3

**Time Complexity:** O(log n)  
**Auxiliary Space:** O(log n)

**Another Iterative Approach to Binary Search**

|  |
| --- |
| def binarySearch(v, To\_Find):      lo = 0      hi = len(v) - 1        # This below check covers all cases , so need to check      # for mid=lo-(hi-lo)/2      while hi - lo > 1:          mid = (hi + lo) // 2          if v[mid] < To\_Find:              lo = mid + 1          else:              hi = mid        if v[lo] == To\_Find:          print("Found At Index", lo)      elif v[hi] == To\_Find:          print("Found At Index", hi)      else:          print("Not Found")      if \_\_name\_\_ == '\_\_main\_\_':      v = [1, 3, 4, 5, 6]        To\_Find = 1      binarySearch(v, To\_Find)        To\_Find = 6      binarySearch(v, To\_Find)        To\_Find = 10      binarySearch(v, To\_Find)    # This code is contributed by Tapesh(tapeshdua420) |

**Output**

Found At Index 0

Found At Index 4

Not Found

**Time Complexity:** O (log n)  
**Auxiliary Space:** O (1)

**Iterative implementation of Binary Search**

|  |
| --- |
| # Python3 code to implement iterative Binary  # Search.    # It returns location of x in given array arr  # if present, else returns -1      def binarySearch(arr, l, r, x):        while l <= r:            mid = l + (r - l) // 2            # Check if x is present at mid          if arr[mid] == x:              return mid            # If x is greater, ignore left half          elif arr[mid] < x:              l = mid + 1            # If x is smaller, ignore right half          else:              r = mid - 1        # If we reach here, then the element      # was not present      return -1      # Driver Code  arr = [2, 3, 4, 10, 40]  x = 10    # Function call  result = binarySearch(arr, 0, len(arr)-1, x)    if result != -1:      print("Element is present at index % d" % result)  else:      print("Element is not present in array") |

**Output**

Element is present at index 3

**Time Complexity:** O(log n)  
**Auxiliary Space:** O(1)

**Algorithmic Paradigm:** [Decrease and Conquer](https://www.geeksforgeeks.org/decrease-and-conquer/).

**Note:**Here we are using

*int mid = low + (high – low)/2;*

Maybe, you wonder why we are calculating the ***middle index***this way, we can simply add the *lower and higher index and divide it by 2.*

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

But if we calculate the***middle index***like this means our code is not 100% correct, it contains bugs.

That is, it fails for larger values of int variables low and high. Specifically, it fails if the sum of low and high is greater than the maximum positive int value(231 – 1).

The sum overflows to a negative value and the value stays negative when divided by 2.   
In java, it throws *ArrayIndexOutOfBoundException.*

*int mid = low + (high – low)/2;*

So it’s better to use it like this. This bug applies equally to merge sort and other divide and conquer algorithms.

# Bubble Sort Algorithm

**Bubble Sort** is the simplest [sorting algorithm](https://www.geeksforgeeks.org/sorting-algorithms/) that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

## ****How does Bubble Sort Work?****

***Input:****arr[] = {5, 1, 4, 2, 8}*

***First Pass:***

* *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
  + *(****5******1****4 2 8 ) –> (****1******5****4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.*
  + *( 1****5******4****2 8 ) –>  ( 1****4******5****2 8 ), Swap since 5 > 4*
  + *( 1 4****5******2****8 ) –>  ( 1 4****2******5****8 ), Swap since 5 > 2*
  + *( 1 4 2****5******8****) –> ( 1 4 2****5******8****), Now, since these elements are already in order (8 > 5), algorithm does not swap them.*

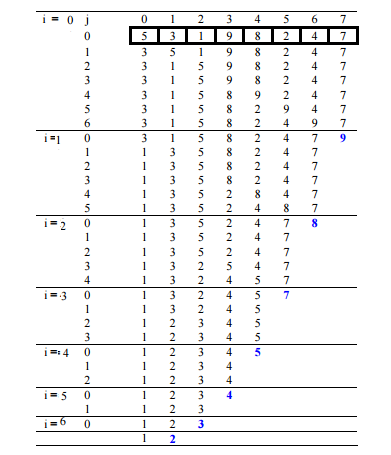
***Second Pass:***

* *Now, during second iteration it should look like this:*
  + *(****1******4****2 5 8 ) –> (****1******4****2 5 8 )*
  + *( 1****4******2****5 8 ) –> ( 1****2******4****5 8 ), Swap since 4 > 2*
  + *( 1 2****4******5****8 ) –> ( 1 2****4******5****8 )*
  + *( 1 2 4****5******8****) –>  ( 1 2 4****5******8****)*

***Third Pass:***

* *Now, the array is already sorted, but our algorithm does not know if it is completed.*
* *The algorithm needs one****whole****pass without****any****swap to know it is sorted.*
  + *(****1******2****4 5 8 ) –> (****1******2****4 5 8 )*
  + *( 1****2******4****5 8 ) –> ( 1****2******4****5 8 )*
  + *( 1 2****4******5****8 ) –> ( 1 2****4******5****8 )*
  + *( 1 2 4****5******8****) –> ( 1 2 4****5******8****)*

***Illustration:***

**

Recommended Problem

Bubble Sort

[Solve Problem](https://practice.geeksforgeeks.org/problems/bubble-sort/1" \o "Permalink to Bubble Sort)

Submission count: 90.6K

Follow the below steps to solve the problem:

* Run a nested for loop to traverse the input array using two variables **i** and **j**, such that 0 ≤ i < n-1 and 0 ≤ j < n-i-1
* If **arr[j]** is greater than **arr[j+1]** then swap these adjacent elements, else move on
* Print the sorted array

Below is the implementation of the above approach:

* C
* C++
* Java
* Python3
* C#
* PHP
* Javascript

|  |
| --- |
| # Python program for implementation of Bubble Sort      def bubbleSort(arr):      n = len(arr)        # Traverse through all array elements      for i in range(n):            # Last i elements are already in place          for j in range(0, n-i-1):                # traverse the array from 0 to n-i-1              # Swap if the element found is greater              # than the next element              if arr[j] > arr[j+1]:                  arr[j], arr[j+1] = arr[j+1], arr[j]      # Driver code to test above  if \_\_name\_\_ == "\_\_main\_\_":    arr = [64, 34, 25, 12, 22, 11, 90]      bubbleSort(arr)      print("Sorted array is:")    for i in range(len(arr)):        print("%d" % arr[i], end=" ") |

**Output**

Sorted array:

1 2 4 5 8

**Time Complexity:** O(N2)  
**Auxiliary Space:** O(1)

## ****Optimized Implementation of Bubble Sort:****

The above function always runs **O(N2)** time even if the array is sorted. It can be optimized by stopping the algorithm if the inner loop didn’t cause any swap.

Below is the implementation for the above approach:

* C
* C++
* Java
* Python3
* C#
* PHP
* Javascript

|  |
| --- |
| # Python3 Optimized implementation  # of Bubble sort    # An optimized version of Bubble Sort  def bubbleSort(arr):      n = len(arr)        # Traverse through all array elements      for i in range(n):          swapped = False            # Last i elements are already          #  in place          for j in range(0, n-i-1):                # traverse the array from 0 to              # n-i-1. Swap if the element              # found is greater than the              # next element              if arr[j] > arr[j+1] :                  arr[j], arr[j+1] = arr[j+1], arr[j]                  swapped = True            # IF no two elements were swapped          # by inner loop, then break          if swapped == False:              break    # Driver code to test above  arr = [64, 34, 25, 12, 22, 11, 90]    bubbleSort(arr)    print ("Sorted array :")  for i in range(len(arr)):      print ("%d" %arr[i],end=" ")    # This code is contributed by Shreyanshi Arun |

**Output**

Sorted array:

1 2 3 4 5 7 8 9

**Time Complexity:**O(N2)  
**Auxiliary Space:** O(1)

## ****Worst Case Analysis for Bubble Sort:****

The **worst-case** condition for bubble sort occurs when elements of the array are arranged in decreasing order.  
In the worst case, the total number of iterations or passes required to sort a given array is **(n-1).**where ‘n’ is a number of elements present in the array.

***At pass 1 :****Number of comparisons = (n-1)  
                     Number of swaps = (n-1)*

***At pass 2 :****Number of comparisons = (n-2)  
                     Number of swaps = (n-2)*

***At pass 3 :****Number of comparisons = (n-3)  
                    Number of swaps = (n-3)****.******.******.******At pass n-1 :****Number of comparisons = 1  
                        Number of swaps = 1*

*Now , calculating total number of comparison required to sort the array  
= (n-1) + (n-2) +  (n-3) + . . . 2 + 1  
= (n-1)\*(n-1+1)/2  { by using sum of N natural Number formula }  
=****n (n-1)/2***

#### **For the Worst case:**

***Total number of swaps = Total number of comparison*** *Total number of comparison (Worst case) =****n(n-1)/2*** *Total number of swaps (Worst case) =****n(n-1)/2***

***Worst and Average Case Time Complexity:****O(N2). The worst case occurs when an array is reverse sorted.****Best Case Time Complexity:****O(N). The best case occurs when an array is already sorted.****Auxiliary Space:****O(1)*

## Recursive Implementation Of Bubble Sort:

*The idea is to place the largest element in its position and keep doing the same for every other element.*

Follow the below steps to solve the problem:

* Place the largest element at its position, this operation makes sure that the first largest element will be placed at the end of the array.
* Recursively call for rest **n – 1** elements with the same operation and place the next greater element at their position.
* The base condition for this recursion call would be, when the number of elements in the array becomes **0** or **1** then, simply return (as they are already sorted).

Below is the implementation of the above approach:

* C++

|  |
| --- |
| //C++ code for recursive bubble sort algorithm  #include <iostream>  using namespace std;  void bubblesort(int arr[], int n)  {      if (n == 0 || n == 1)      {          return;      }      for (int i = 0; i < n - 1; i++)      {          if (arr[i] > arr[i + 1])          {              swap(arr[i], arr[i + 1]);          }      }      bubblesort(arr, n - 1);  }  int main()  {      int arr[5] = {2, 5, 1, 6, 9};      bubblesort(arr, 5);      for (int i = 0; i < 5; i++)      {          cout << arr[i] << " ";      }      return 0;  }  //code contributed by pragatikohli |

**Output**

1 2 5 6 9

## ****What is the Boundary Case for Bubble sort?****

Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid O(N2) time complexity.

## Does sorting happen in place in Bubble sort?

Yes, Bubble sort performs swapping of adjacent pairs without the use of any major data structure. Hence Bubble sort algorithm is an in-place algorithm.

## Is the Bubble sort algorithm stable?

Yes, the bubble sort algorithm is stable.

## Where is the Bubble sort algorithm used?

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.   
In computer graphics, it is popular for its capability to detect a tiny error (like a swap of just two elements) in almost-sorted arrays and fix it with just linear   
complexity (2n).

# Insertion Sort

**Insertion sort** is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

### **Characteristics of Insertion Sort:**

* This algorithm is one of the simplest algorithm with simple implementation
* Basically, Insertion sort is efficient for small data values
* Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

### Working of Insertion Sort algorithm:

*Consider an example: arr[]: {12, 11, 13, 5, 6}*

| 12 | 11 | 13 | 5 | 6 |
| --- | --- | --- | --- | --- |

***First Pass:***

* *Initially, the first two elements of the array are compared in insertion sort.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **12** | **11** | 13 | 5 | 6 |

* *Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.*
* *So, for now 11 is stored in a sorted sub-array.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **11** | **12** | 13 | 5 | 6 |

***Second Pass:***

* *Now, move to the next two elements and compare them*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **12** | **13** | 5 | 6 |

* *Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11*

***Third Pass:***

* *Now, two elements are present in the sorted sub-array which are****11****and****12***
* *Moving forward to the next two elements which are 13 and 5*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **13** | **5** | 6 |

* *Both 5 and 13 are not present at their correct place so swap them*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **5** | **13** | 6 |

* *After swapping, elements 12 and 5 are not sorted, thus swap again*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **5** | **12** | 13 | 6 |

* *Here, again 11 and 5 are not sorted, hence swap again*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **11** | 12 | 13 | 6 |

* *here, it is at its correct position*

***Fourth Pass:***

* *Now, the elements which are present in the sorted sub-array are****5, 11****and****12***
* *Moving to the next two elements 13 and 6*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | **13** | **6** |

* *Clearly, they are not sorted, thus perform swap between both*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | **6** | **13** |

* *Now, 6 is smaller than 12, hence, swap again*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | **6** | **12** | 13 |

* *Here, also swapping makes 11 and 6 unsorted hence, swap again*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **6** | **11** | 12 | 13 |

* *Finally, the array is completely sorted.*

***Illustrations:***

**

### **Insertion Sort Algorithm**

To sort an array of size N in ascending order:

* Iterate from arr[1] to arr[N] over the array.
* Compare the current element (key) to its predecessor.
* If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Recommended Problem

Insertion Sort

[Solve Problem](https://practice.geeksforgeeks.org/problems/insertion-sort/1" \o "Permalink to Insertion Sort)

|  |
| --- |
| # Python program for implementation of Insertion Sort    # Function to do insertion sort  def insertionSort(arr):        # Traverse through 1 to len(arr)      for i in range(1, len(arr)):            key = arr[i]            # Move elements of arr[0..i-1], that are          # greater than key, to one position ahead          # of their current position          j = i-1          while j >= 0 and key < arr[j] :                  arr[j + 1] = arr[j]                  j -= 1          arr[j + 1] = key      # Driver code to test above  arr = [12, 11, 13, 5, 6]  insertionSort(arr)  for i in range(len(arr)):      print ("% d" % arr[i])    # This code is contributed by Mohit Kumra |

**Output**

5 6 11 12 13

**Time Complexity:** O(N^2)   
**Auxiliary Space:**O(1)

### **What are the Boundary Cases of Insertion Sort algorithm?**

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

### **What are the Algorithmic Paradigm of Insertion Sort algorithm?**

Insertion Sort algorithm follows incremental approach.

### **Is Insertion Sort an in-place sorting algorithm?**

Yes, insertion sort is an in-place sorting algorithm.

### **Is Insertion Sort a stable algorithm?**

Yes, insertion sort is a stable sorting algorithm.

### **When is the Insertion Sort algorithm used?**

Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

### **What is Binary Insertion Sort?**

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes O(i) (at ith iteration) in worst case. We can reduce it to O(logi) by using binary search. The algorithm, as a whole, still has a running worst case running time of O(n^2) because of the series of swaps required for each insertion. Refer [this](https://www.geeksforgeeks.org/binary-insertion-sort/) for implementation.

**Selection Sort Algorithm**

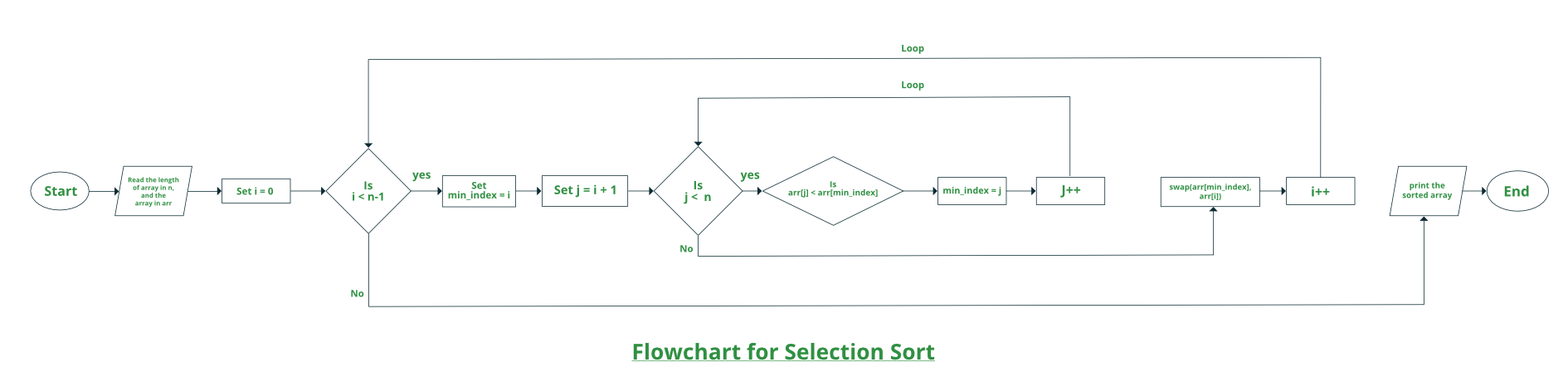
The **selection sort algorithm** sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

* The subarray which already sorted.
* The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

**Flowchart of the Selection Sort:**

**  
How selection sort works?**

*Lets consider the following array as an example:****arr[] = {64, 25, 12, 22, 11}***

***First pass:***

* *For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where****64****is stored presently, after traversing whole array it is clear that****11****is the lowest value.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **64** | 25 | 12 | 22 | 11 |

* *Thus, replace 64 with 11. After one iteration****11****, which happens to be the least value in the array, tends to appear in the first position of the sorted list.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **11** | 25 | 12 | 22 | 64 |

***Second Pass:***

* *For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **25** | 12 | 22 | 64 |

* *After traversing, we found that****12****is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **12** | 25 | 22 | 64 |

***Third Pass:***

* *Now, for third place, where****25****is present again traverse the rest of the array and find the third least value present in the array.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **25** | 22 | 64 |

* *While traversing,****22****came out to be the third least value and it should appear at the third place in the array, thus swap****22****with element present at third position.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **22** | 25 | 64 |

***Fourth pass:***

* *Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array*
* *As****25****is the 4th lowest value hence, it will place at the fourth position.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 22 | **25** | 64 |

***Fifth Pass:***

* *At last the largest value present in the array automatically get placed at the last position in the array*
* *The resulted array is the sorted array.*

| 11 | 12 | 22 | **25** | 64 |
| --- | --- | --- | --- | --- |

Recommended Problem

Selection Sort

Follow the below steps to solve the problem:

* Initialize minimum value(**min\_idx**) to location 0.
* Traverse the array to find the minimum element in the array.
* While traversing if any element smaller than **min\_idx**is found then swap both the values.
* Then, increment **min\_idx** to point to the next element.
* Repeat until the array is sorted.

Below is the implementation of the above approach:

|  |
| --- |
| # Python program for implementation of Selection  # Sort  import sys  A = [64, 25, 12, 22, 11]    # Traverse through all array elements  for i in range(len(A)):        # Find the minimum element in remaining      # unsorted array      min\_idx = i      for j in range(i+1, len(A)):          if A[min\_idx] > A[j]:              min\_idx = j        # Swap the found minimum element with      # the first element      A[i], A[min\_idx] = A[min\_idx], A[i]    # Driver code to test above  print ("Sorted array")  for i in range(len(A)):      print("%d" %A[i],end=" ") |

**Output**

Sorted array:

11 12 22 25 64

**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)

**Auxiliary Space:** O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory write is a costly operation.

**Merge Sort Algorithm**

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

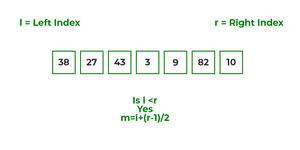
**Merge Sort Working Process:**

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

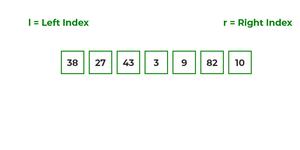
**Illustration:**

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

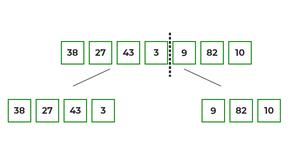
* *At first, check if the left index of array is less than the right index, if yes then calculate its mid point*

**

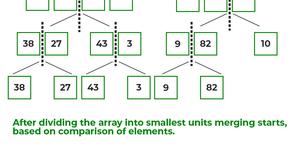
* *Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.*
* *Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.*

**

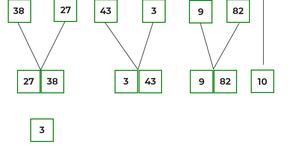
* *Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.*

**

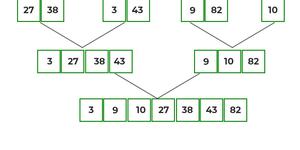
* *Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.*

**

* *After dividing the array into smallest units, start merging the elements again based on comparison of size of elements*
* *Firstly, compare the element for each list and then combine them into another list in a sorted manner.*

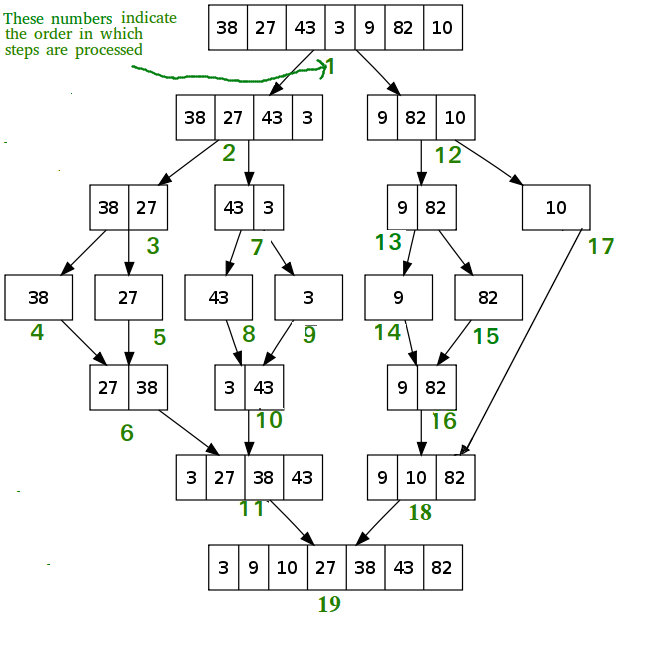
**

* *After the final merging, the list looks like this:*

**

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



*Recursive steps of merge sort*

**Algorithm:**

*step 1: start*

*step 2: declare array and left, right, mid variable*

*step 3: perform merge function.  
    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*

Follow the steps below the solve the problem:

MergeSort(arr[], l,  r)  
If r > l

* Find the middle point to divide the array into two halves:
  + middle m = l + (r – l)/2
* Call mergeSort for first half:
  + Call mergeSort(arr, l, m)
* Call mergeSort for second half:
  + Call mergeSort(arr, m + 1, r)
* Merge the two halves sorted in steps 2 and 3:
  + Call merge(arr, l, m, r)

Below is the implementation of the above approach:

|  |
| --- |
| # Python program for implementation of MergeSort  def mergeSort(arr):      if len(arr) > 1:             # Finding the mid of the array          mid = len(arr)//2            # Dividing the array elements          L = arr[:mid]            # into 2 halves          R = arr[mid:]            # Sorting the first half          mergeSort(L)            # Sorting the second half          mergeSort(R)            i = j = k = 0            # Copy data to temp arrays L[] and R[]          while i < len(L) and j < len(R):              if L[i] < R[j]:                  arr[k] = L[i]                  i += 1              else:                  arr[k] = R[j]                  j += 1              k += 1            # Checking if any element was left          while i < len(L):              arr[k] = L[i]              i += 1              k += 1            while j < len(R):              arr[k] = R[j]              j += 1              k += 1    # Code to print the list      def printList(arr):      for i in range(len(arr)):          print(arr[i], end=" ")      print()      # Driver Code  if \_\_name\_\_ == '\_\_main\_\_':      arr = [12, 11, 13, 5, 6, 7]      print("Given array is", end="\n")      printList(arr)      mergeSort(arr)      print("Sorted array is: ", end="\n")      printList(arr)    # This code is contributed by Mayank Khanna |

**Output**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:**O(N log(N)),  Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

*T(n) = 2T(n/2) + θ(n)*

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

**Auxiliary Space:** O(n), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

# QuickSort

Like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), **QuickSort**is a[Divide and Conquer algorithm](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/). It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick the first element as a pivot.
* Always pick the last element as a pivot (implemented below)
* Pick a random element as a pivot.
* Pick median as the pivot.

The key process in **quickSort**is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



**Partition Algorithm:**

There can be many ways to do partition, following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

**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 \*/*

*pi = partition(arr, low, high);*

*quickSort(arr, low, pi – 1);  // Before pi*

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

*}*

*}*

**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 and indicates the   
// right position of pivot found so far*

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

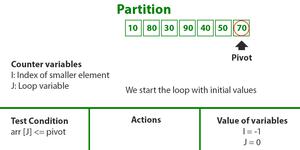
*// If current element is smaller than the pivot  
if (arr[j] < pivot){  
i++;    // increment index of smaller element  
 swap arr[i] and arr[j]  
     }  
 }*

*swap arr[i + 1] and arr[high])  
return (i + 1)  
}*

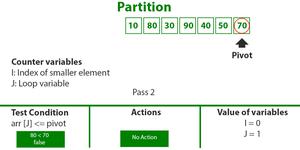
**Illustration of partition() :**

*Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}*

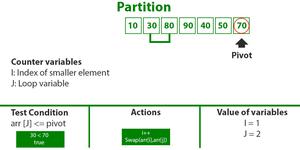
* *Indexes:  0   1   2   3   4   5   6*
* *low = 0, high =  6, pivot = arr[h] = 70*
* *Initialize index of smaller element,****i = -1***

**

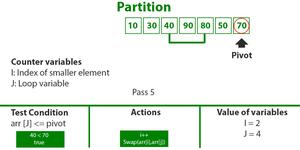
* *Traverse elements from j = low to high-1*
  + ***j = 0****: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
  + ***i = 0***
* *arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same*
* ***j = 1****: Since arr[j] > pivot, do nothing*

**

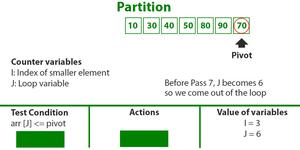
* ***j = 2****: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
* ***i = 1***
* *arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30*

**

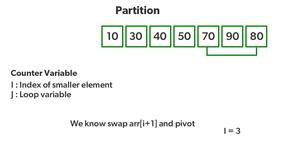
* ***j = 3****: Since arr[j] > pivot, do nothing // No change in i and arr[]*
* ***j = 4****: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
* ***i = 2***
* *arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped*

**

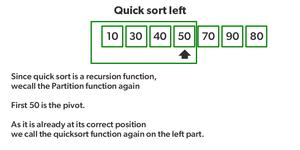
* ***j = 5****: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]*
* ***i = 3***
* *arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped*

**

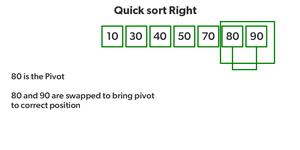
* *We come out of loop because j is now equal to high-1.*
* ***Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)***
* *arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped*

**

* *Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.*
* *Since quick sort is a recursive function, we call the partition function again at left and right partitions*

**

* *Again call function at right part and swap 80 and 90*

**

**Implementation:**   
Following are the implementations of QuickSort:

* C++14
* Java
* Python3
* C#
* Javascript

|  |
| --- |
| # Python3 implementation of QuickSort      # Function to find the partition position  def partition(array, low, high):      # Choose the rightmost element as pivot    pivot = array[high]      # Pointer for greater element    i = low - 1      # Traverse through all elements    # compare each element with pivot    for j in range(low, high):      if array[j] <= pivot:        # If element smaller than pivot is found        # swap it with the greater element pointed by i        i = i + 1          # Swapping element at i with element at j        (array[i], array[j]) = (array[j], array[i])      # Swap the pivot element with the greater element specified by i    (array[i + 1], array[high]) = (array[high], array[i + 1])      # Return the position from where partition is done    return i + 1    # Function to perform quicksort  def quick\_sort(array, low, high):    if low < high:        # Find pivot element such that      # element smaller than pivot are on the left      # element greater than pivot are on the right      pi = partition(array, low, high)        # Recursive call on the left of pivot      quick\_sort(array, low, pi - 1)        # Recursive call on the right of pivot      quick\_sort(array, pi + 1, high)        # Driver code  array = [ 10, 7, 8, 9, 1, 5]  quick\_sort(array, 0, len(array) - 1)    print(f'Sorted array: {array}')    # This code is contributed by Adnan Aliakbar |

**Output**

Sorted array:

1 5 7 8 9 10

### Implementation of QuickSort using the first element as a pivot:

|  |
| --- |
| # Python3 implementation of QuickSort      # Function to find the partition position  def partition(arr, l, h):    low, high = l, h    if l != h and l < h:      # Choose the leftmost element as pivot      pivot = arr[l]      low = low+1      # Traverse through all elements      # compare each element with pivot      while low <= high:        if arr[high] < pivot and arr[low] > pivot:          arr[high], arr[low] = arr[low], arr[high]        if not arr[low] > pivot:          low += 1        if not arr[high] < pivot:          high -= 1    arr[l], arr[high] = arr[high], arr[l]    # Return the position from where partition is done    return high    # Function to perform quicksort  def quick\_sort(array, low, high):    if low < high:          # Find pivot element such that        # element smaller than pivot are on the left        # element greater than pivot are on the right        pi = partition(array, low, high)          # Recursive call on the left of pivot        quick\_sort(array, low, pi - 1)          # Recursive call on the right of pivot        quick\_sort(array, pi + 1, high)        # Driver code  array = [ 1, 7, 8, 9, 1, 2]  quick\_sort(array, 0, len(array) - 1)    print(f'Sorted array: {array}')    # This code is contributed by Adnan Aliakbar |

**Output**

Before Sorting

4 2 8 3 1 5 7 11 6

After Sorting

1 2 3 4 5 6 7 8 11

### **Analysis of QuickSort**

Time taken by QuickSort, in general, can be written as follows.

*T(n) = T(k) + T(n-k-1) + (n)*

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.   
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

**Worst Case:**   
The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.

*T(n) = T(0) + T(n-1) + (n)which is equivalent to  T(n) = T(n-1) + (n)*

**The solution to the above recurrence is  (n2).**

**Best Case:**  
The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

*T(n) = 2T(n/2) + (n)*

**The solution for the above recurrence is (nLogn). It can be solved using case 2 of**[**Master Theorem**](http://en.wikipedia.org/wiki/Master_theorem)**.**

**Average Case:**   
To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/).   
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

*T(n) = T(n/9) + T(9n/10) + (n)*

**The solution of above recurrence is also O(nLogn):**

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/) and [Heap Sort](https://www.geeksforgeeks.org/heap-sort/), QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**Radix Sort**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 09 Aug, 2022

 Read

 Discuss

The [lower bound for the Comparison based sorting algorithm](https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/) (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than ***nLogn***. [Counting sort](https://www.geeksforgeeks.org/counting-sort/) is a linear time sorting algorithm that sort in O(n+k) time when elements are in the range from 1 to k.

**What if the elements are in the** **range from 1 to n2?**

We can’t use counting sort because counting sort will take O(n2) which is worse than comparison-based sorting algorithms. Can we sort such an array in linear time?

[Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

**The Radix Sort Algorithm**

Do the following for each digit I where I varies from the least significant digit to the most significant digit. Here we will be sorting the input array using counting sort (or any stable sort) according to the i’th digit.

**Example:**

*Original, unsorted list: 170, 45, 75, 90, 802, 24, 2, 66 Sorting by least significant digit (1s place) gives: [\*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.] 170, 90, 802, 2, 24, 45, 75, 66 Sorting by next digit (10s place) gives: [\*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.] 802, 2, 24, 45, 66, 170, 75, 90 Sorting by the most significant digit (100s place) gives: 2, 24, 45, 66, 75, 90, 170, 802*

**What is the running time of Radix Sort?**

Let there be d digits in input integers. Radix Sort takes O(d\*(n+b)) time where b is the base for representing numbers, for example, for the decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be O(logb(k)). So overall time complexity is O((n+b) \* logb(k)). Which looks more than the time complexity of comparison-based sorting algorithms for a large k. Let us first limit k. Let k <= nc where c is a constant. In that case, the complexity becomes O(nLogb(n)). But it still doesn’t beat comparison-based sorting algorithms.   
What if we make the value of b larger? What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with a range from 1 to nc if the numbers are represented in base n (or every digit takes log2(n) bits).

**Applications of Radix Sort:**

* In a typical computer, which is a sequential random-access machine, where the records are keyed by multiple fields radix sort is used. For eg., you want to sort on three keys month, day and year. You could compare two records on year, then on a tie on month and finally on the date. Alternatively, sorting the data three times using Radix sort first on the date, then on month, and finally on year could be used.
* It was used in card sorting machines with 80 columns, and in each column, the machine could punch a hole only in 12 places. The sorter was then programmed to sort the cards, depending upon which place the card had been punched. This was then used by the operator to collect the cards which had the 1st row punched, followed by the 2nd row, and so on.

**Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?**

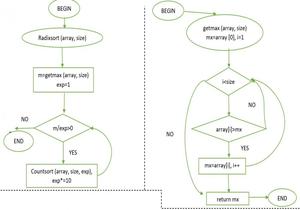
If we have log2n bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.

**Key points about Radix Sort:**

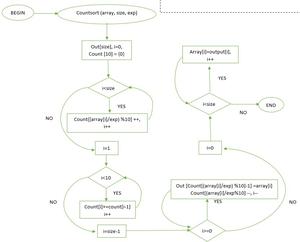
Some key points about radix sort are given here

1. It makes assumptions about the data like the data must be between a range of elements.
2. Input array must have the elements with the same radix and width.
3. Radix sort works on sorting based on an individual digit or letter position.
4. We must start sorting from the rightmost position and use a stable algorithm at each position.
5. Radix sort is not an in-place algorithm as it uses a temporary count array.

**FLowcharts**



*Flowchart radix sort, getmax*



*Flowchart countsort*

**Implementation:**Radix Sort

Recommended Problem

Radix Sort

[Sorting](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Sorting&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/radix-sort/1" \o "Permalink to Radix Sort)

Submission count: 1.2K

Following is a simple implementation of Radix Sort. For simplicity, the value of d is assumed to be 10. We recommend you to see [Counting Sort](https://www.geeksforgeeks.org/counting-sort/) for details of countSort() function in the below code.

|  |
| --- |
| # Python program for implementation of Radix Sort  # A function to do counting sort of arr[] according to  # the digit represented by exp.    def countingSort(arr, exp1):        n = len(arr)        # The output array elements that will have sorted arr      output = [0] \* (n)        # initialize count array as 0      count = [0] \* (10)        # Store count of occurrences in count[]      for i in range(0, n):          index = arr[i] // exp1          count[index % 10] += 1        # Change count[i] so that count[i] now contains actual      # position of this digit in output array      for i in range(1, 10):          count[i] += count[i - 1]        # Build the output array      i = n - 1      while i >= 0:          index = arr[i] // exp1          output[count[index % 10] - 1] = arr[i]          count[index % 10] -= 1          i -= 1        # Copying the output array to arr[],      # so that arr now contains sorted numbers      i = 0      for i in range(0, len(arr)):          arr[i] = output[i]    # Method to do Radix Sort  def radixSort(arr):        # Find the maximum number to know number of digits      max1 = max(arr)        # Do counting sort for every digit. Note that instead      # of passing digit number, exp is passed. exp is 10^i      # where i is current digit number      exp = 1      while max1 / exp >= 1:          countingSort(arr, exp)          exp \*= 10      # Driver code  arr = [170, 45, 75, 90, 802, 24, 2, 66]    # Function Call  radixSort(arr)    for i in range(len(arr)):      print(arr[i],end=" ")    # This code is contributed by Mohit Kumra  # Edited by Patrick Gallagher |

**Output**

2 24 45 66 75 90 170 802

Following is another way of the implementation of the radix sort while using the bucket sort technique, it might not look simple while having a look at the code but if you give it a shot it’s quite easy, [one must know Bucket Sort to deeper depth](https://www.geeksforgeeks.org/bucket-sort-2).

**Output**

6 12 25 161 415 573

Time complexities remain the same as in the first method, it’s just the implementation through another method.

**Radix Sort on Strings:** Radix sort is mostly used to sort the numerical values or the real values, but it can be modified to sort the string values in lexicographical order. It follows the same procedure as used for numerical values.

Please refer [this IDE link](https://ide.geeksforgeeks.org/79c17ad3-f9be-46ee-9041-5729fc40cd1a) for the implementation of the same.

**Output**

Input:[BCDEF, dbaqc, abcde, bbbbb]

Output:[abcde, bbbbb, BCDEF, dbaqc]

**Snapshots:**

