Heap Sort Algorithm

In this article, we will discuss the Heapsort Algorithm. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

* Build a heap H, using the elements of array.
* Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap.**

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is heap sort?

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Now, let's see the algorithm of heap sort.

Algorithm

1. HeapSort(arr)
2. BuildMaxHeap(arr)
3. for i = length(arr) to 2
4. swap arr[1] with arr[i]
5. heap\_size[arr] = heap\_size[arr] ? 1
6. MaxHeapify(arr,1)
7. End

**BuildMaxHeap(arr)**

1. BuildMaxHeap(arr)
2. heap\_size(arr) = length(arr)
3. for i = length(arr)/2 to 1
4. MaxHeapify(arr,i)
5. End

**MaxHeapify(arr,i)**

1. MaxHeapify(arr,i)
2. L = left(i)
3. R = right(i)
4. if L ? heap\_size[arr] and arr[L] **>** arr[i]
5. largest = L
6. else
7. largest = i
8. if R ? heap\_size[arr] and arr[R] **>** arr[largest]
9. largest = R
10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End

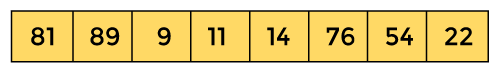
Working of Heap sort Algorithm

Now, let's see the working of the Heapsort Algorithm.

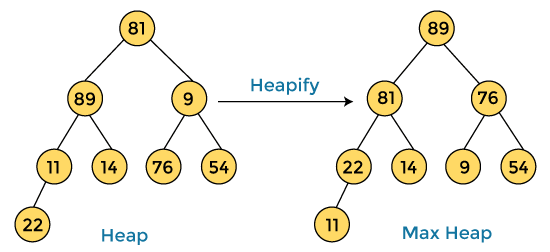
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

* The first step includes the creation of a heap by adjusting the elements of the array.
* After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

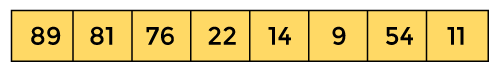
Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.



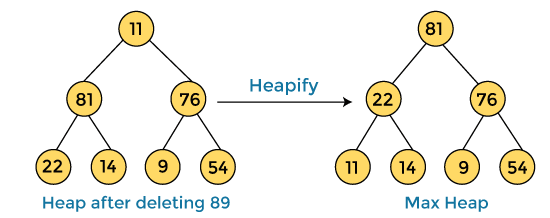
First, we have to construct a heap from the given array and convert it into max heap.



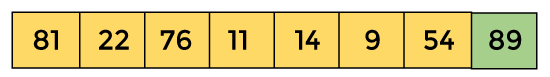
After converting the given heap into max heap, the array elements are -



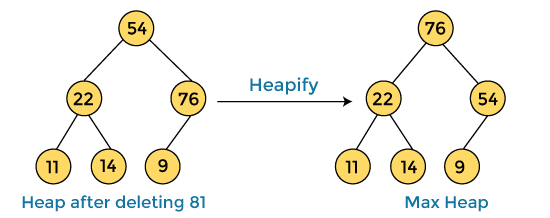
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



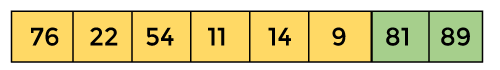
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



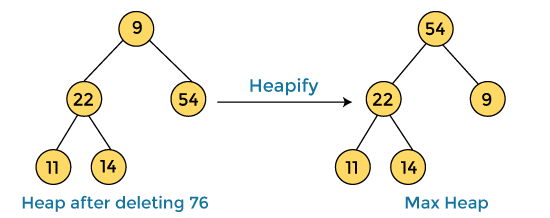
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



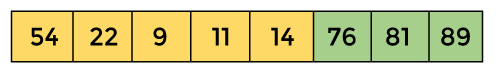
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -



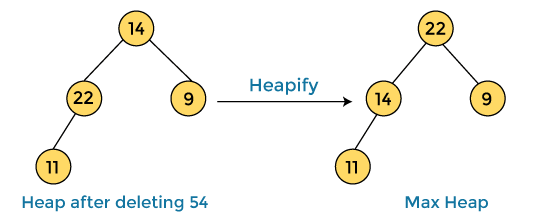
In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



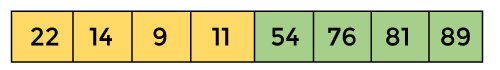
After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -



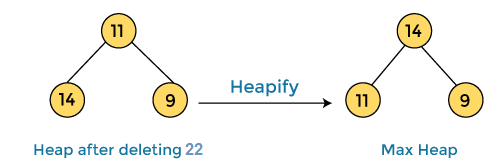
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



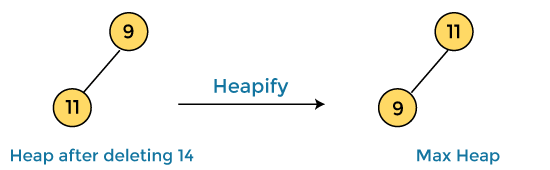
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



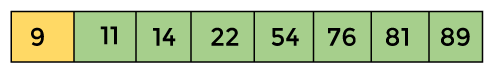
After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



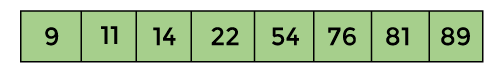
After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are -



Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n logn) |
| **Average Case** | O(n log n) |
| **Worst Case** | O(n log n) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | N0 |

* The space complexity of Heap sort is O(1).

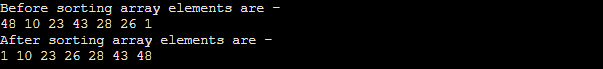
Implementation of Heapsort

Now, let's see the programs of Heap sort in different programming languages.

**Program:** Write a program to implement heap sort in C language.

1. #include **<stdio.h>**
2. /\* function to heapify a subtree. Here 'i' is the
3. index of root node in array a[], and 'n' is the size of heap. \*/
4. void heapify(int a[], int n, int i)
5. {
6. int largest = i; // Initialize largest as root
7. int left = 2 \* i + 1; // left child
8. int right = 2 \* i + 2; // right child
9. // If left child is larger than root
10. if (left **<** **n** && a[left] **>** a[largest])
11. largest = left;
12. // If right child is larger than root
13. if (right **<** **n** && a[right] **>** a[largest])
14. largest = right;
15. // If root is not largest
16. if (largest != i) {
17. // swap a[i] with a[largest]
18. int temp = a[i];
19. a[i] = a[largest];
20. a[largest] = temp;
22. heapify(a, n, largest);
23. }
24. }
25. /\*Function to implement the heap sort\*/
26. void heapSort(int a[], int n)
27. {
28. for (int i = n / 2 - 1; i **>**= 0; i--)
29. heapify(a, n, i);
30. // One by one extract an element from heap
31. for (int i = n - 1; i **>**= 0; i--) {
32. /\* Move current root element to end\*/
33. // swap a[0] with a[i]
34. int temp = a[0];
35. a[0] = a[i];
36. a[i] = temp;
38. heapify(a, i, 0);
39. }
40. }
41. /\* function to print the array elements \*/
42. void printArr(int arr[], int n)
43. {
44. for (int i = 0; i **<** **n**; ++i)
45. {
46. printf("%d", arr[i]);
47. printf(" ");
48. }
50. }
51. int main()
52. {
53. int a[] = {48, 10, 23, 43, 28, 26, 1};
54. int n = sizeof(a) / sizeof(a[0]);
55. printf("Before sorting array elements are - \n");
56. printArr(a, n);
57. heapSort(a, n);
58. printf("\nAfter sorting array elements are - \n");
59. printArr(a, n);
60. return 0;
61. }

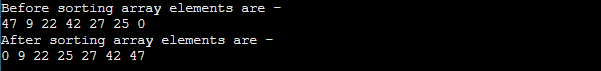
**Output**



**Program:** Write a program to implement heap sort in C++.

1. #include **<iostream>**
2. using namespace std;
3. /\* function to heapify a subtree. Here 'i' is the
4. index of root node in array a[], and 'n' is the size of heap. \*/
5. void heapify(int a[], int n, int i)
6. {
7. int largest = i; // Initialize largest as root
8. int left = 2 \* i + 1; // left child
9. int right = 2 \* i + 2; // right child
10. // If left child is larger than root
11. if (left **<** **n** && a[left] **>** a[largest])
12. largest = left;
13. // If right child is larger than root
14. if (right **<** **n** && a[right] **>** a[largest])
15. largest = right;
16. // If root is not largest
17. if (largest != i) {
18. // swap a[i] with a[largest]
19. int temp = a[i];
20. a[i] = a[largest];
21. a[largest] = temp;
23. heapify(a, n, largest);
24. }
25. }
26. /\*Function to implement the heap sort\*/
27. void heapSort(int a[], int n)
28. {
30. for (int i = n / 2 - 1; i **>**= 0; i--)
31. heapify(a, n, i);
32. // One by one extract an element from heap
33. for (int i = n - 1; i **>**= 0; i--) {
34. /\* Move current root element to end\*/
35. // swap a[0] with a[i]
36. int temp = a[0];
37. a[0] = a[i];
38. a[i] = temp;
40. heapify(a, i, 0);
41. }
42. }
43. /\* function to print the array elements \*/
44. void printArr(int a[], int n)
45. {
46. for (int i = 0; i **<** **n**; ++i)
47. {
48. cout**<<a**[i]**<<**" ";
49. }
51. }
52. int main()
53. {
54. int a[] = {47, 9, 22, 42, 27, 25, 0};
55. int n = sizeof(a) / sizeof(a[0]);
56. cout**<<**"Before sorting array elements are - \n";
57. printArr(a, n);
58. heapSort(a, n);
59. cout**<<**"\nAfter sorting array elements are - \n";
60. printArr(a, n);
61. return 0;
62. }

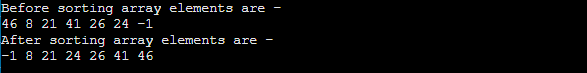
**Output**



**Program:** Write a program to implement heap sort in C#.

1. using System;
2. class HeapSort {
3. /\* function to heapify a subtree. Here 'i' is the
4. index of root node in array a[], and 'n' is the size of heap. \*/
5. static void heapify(int[] a, int n, int i)
6. {
7. int largest = i; // Initialize largest as root
8. int left = 2 \* i + 1; // left child
9. int right = 2 \* i + 2; // right child
10. // If left child is larger than root
11. if (left **<** **n** && a[left] **>** a[largest])
12. largest = left;
13. // If right child is larger than root
14. if (right **<** **n** && a[right] **>** a[largest])
15. largest = right;
16. // If root is not largest
17. if (largest != i) {
18. // swap a[i] with a[largest]
19. int temp = a[i];
20. a[i] = a[largest];
21. a[largest] = temp;
23. heapify(a, n, largest);
24. }
25. }
26. /\*Function to implement the heap sort\*/
27. static void heapSort(int[] a, int n)
28. {
29. for (int i = n / 2 - 1; i **>**= 0; i--)
30. heapify(a, n, i);
32. // One by one extract an element from heap
33. for (int i = n - 1; i **>**= 0; i--) {
34. /\* Move current root element to end\*/
35. // swap a[0] with a[i]
36. int temp = a[0];
37. a[0] = a[i];
38. a[i] = temp;
40. heapify(a, i, 0);
41. }
42. }
43. /\* function to print the array elements \*/
44. static void printArr(int[] a, int n)
45. {
46. for (int i = 0; i **<** **n**; ++i)
47. Console.Write(a[i] + " ");
48. }
49. static void Main()
50. {
51. int[] a = {46, 8, 21, 41, 26, 24, -1};
52. int n = a.Length;
53. Console.Write("Before sorting array elements are - \n");
54. printArr(a, n);
55. heapSort(a, n);
56. Console.Write("\nAfter sorting array elements are - \n");
57. printArr(a, n);
58. }
59. }

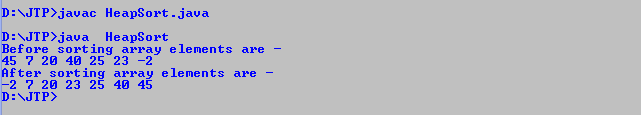
**Output**



**Program:** Write a program to implement heap sort in Java.

1. class HeapSort
2. {
3. /\* function to heapify a subtree. Here 'i' is the
4. index of root node in array a[], and 'n' is the size of heap. \*/
5. static void heapify(int a[], int n, int i)
6. {
7. int largest = i; // Initialize largest as root
8. int left = 2 \* i + 1; // left child
9. int right = 2 \* i + 2; // right child
10. // If left child is larger than root
11. if (left **<** **n** && a[left] **>** a[largest])
12. largest = left;
13. // If right child is larger than root
14. if (right **<** **n** && a[right] **>** a[largest])
15. largest = right;
16. // If root is not largest
17. if (largest != i) {
18. // swap a[i] with a[largest]
19. int temp = a[i];
20. a[i] = a[largest];
21. a[largest] = temp;
23. heapify(a, n, largest);
24. }
25. }
26. /\*Function to implement the heap sort\*/
27. static void heapSort(int a[], int n)
28. {
29. for (int i = n / 2 - 1; i **>**= 0; i--)
30. heapify(a, n, i);
32. // One by one extract an element from heap
33. for (int i = n - 1; i **>**= 0; i--) {
34. /\* Move current root element to end\*/
35. // swap a[0] with a[i]
36. int temp = a[0];
37. a[0] = a[i];
38. a[i] = temp;
40. heapify(a, i, 0);
41. }
42. }
43. /\* function to print the array elements \*/
44. static void printArr(int a[], int n)
45. {
46. for (int i = 0; i **<** **n**; ++i)
47. System.out.print(a[i] + " ");
48. }
49. public static void main(String args[])
50. {
51. int a[] = {45, 7, 20, 40, 25, 23, -2};
52. int n = a.length;
53. System.out.print("Before sorting array elements are - \n");
54. printArr(a, n);
55. heapSort(a, n);
56. System.out.print("\nAfter sorting array elements are - \n");
57. printArr(a, n);
58. }
59. }

**Output**



So, that's all about the article. Hope the article will be helpful and informative to you.

Insertion Sort Algorithm

In this article, we will discuss the Insertion sort Algorithm. The working procedure of insertion sort is also simple. This article will be very helpful and interesting to students as they might face insertion sort as a question in their examinations. So, it is important to discuss the topic.

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **O(n2)**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

* Simple implementation
* Efficient for small data sets
* Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

**Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.

**Step2 -** Pick the next element, and store it separately in a **key.**

**Step3 -** Now, compare the **key** with all elements in the sorted array.

**Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

**Step 5 -** Insert the value.

**Step 6 -** Repeat until the array is sorted.

Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

Insertion Sort Algorithm

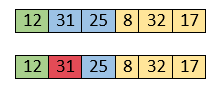
Initially, the first two elements are compared in insertion sort.

Insertion Sort Algorithm

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

Insertion Sort Algorithm

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

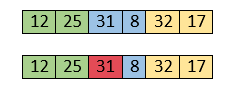


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

Insertion Sort Algorithm

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



Both 31 and 8 are not sorted. So, swap them.

Insertion Sort Algorithm

After swapping, elements 25 and 8 are unsorted.

Insertion Sort Algorithm

So, swap them.

Insertion Sort Algorithm

Now, elements 12 and 8 are unsorted.

Insertion Sort Algorithm

So, swap them too.

Insertion Sort Algorithm

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

Insertion Sort Algorithm

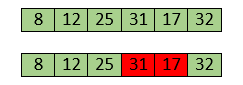
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Insertion Sort Algorithm

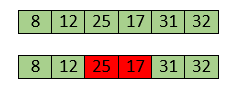
Move to the next elements that are 32 and 17.

Insertion Sort Algorithm

17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

Insertion Sort Algorithm

Now, the array is completely sorted.

Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n2)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n2)**.

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | YES |

* The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

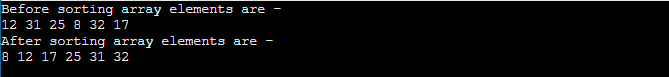
Implementation of insertion sort

Now, let's see the programs of insertion sort in different programming languages.

**Program:** Write a program to implement insertion sort in C language.

1. #include <stdio.h>
3. **void** insert(**int** a[], **int** n) /\* function to sort an aay with insertion sort \*/
4. {
5. **int** i, j, temp;
6. **for** (i = 1; i < n; i++) {
7. temp = a[i];
8. j = i - 1;
10. **while**(j>=0 && temp <= a[j])  /\* Move the elements greater than temp to one position ahead from their current position\*/
11. {
12. a[j+1] = a[j];
13. j = j-1;
14. }
15. a[j+1] = temp;
16. }
17. }
19. **void** printArr(**int** a[], **int** n) /\* function to print the array \*/
20. {
21. **int** i;
22. **for** (i = 0; i < n; i++)
23. printf("%d ", a[i]);
24. }
26. **int** main()
27. {
28. **int** a[] = { 12, 31, 25, 8, 32, 17 };
29. **int** n = **sizeof**(a) / **sizeof**(a[0]);
30. printf("Before sorting array elements are - \n");
31. printArr(a, n);
32. insert(a, n);
33. printf("\nAfter sorting array elements are - \n");
34. printArr(a, n);
36. **return** 0;
37. }

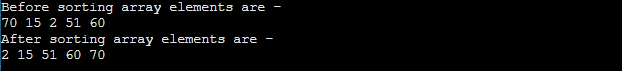
**Output:**



**Program:** Write a program to implement insertion sort in python.

1. def insertionSort(a): # Function to implement insertion sort
2. **for** i in range(1, len(a)):
3. temp = a[i]
5. # Move the elements greater than temp to one position
6. #ahead from their current position
7. j = i-1
8. **while** j >= 0 and temp < a[j] :
9. a[j + 1] = a[j]
10. j = j-1
11. a[j + 1] = temp
13. def printArr(a): # function to print the array
15. **for** i in range(len(a)):
16. print (a[i], end = " ")
18. a = [70, 15, 2, 51, 60]
19. print("Before sorting array elements are - ")
20. printArr(a)
21. insertionSort(a)
22. print("\nAfter sorting array elements are - ")
23. printArr(a)

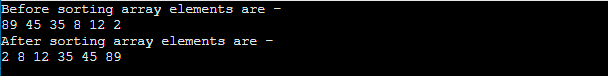
**Output:**



**Program:** Write a program to implement insertion sort in C++ language.

1. #include <iostream>
2. **using** **namespace** std;
4. **void** insert(**int** a[], **int** n) /\* function to sort an aay with insertion sort \*/
5. {
6. **int** i, j, temp;
7. **for** (i = 1; i < n; i++) {
8. temp = a[i];
9. j = i - 1;
11. **while**(j>=0 && temp <= a[j])  /\* Move the elements greater than temp to one position ahead from their current position\*/
12. {
13. a[j+1] = a[j];
14. j = j-1;
15. }
16. a[j+1] = temp;
17. }
18. }
20. **void** printArr(**int** a[], **int** n) /\* function to print the array \*/
21. {
22. **int** i;
23. **for** (i = 0; i < n; i++)
24. cout << a[i] <<" ";
25. }
27. **int** main()
28. {
29. **int** a[] = { 89, 45, 35, 8, 12, 2 };
30. **int** n = **sizeof**(a) / **sizeof**(a[0]);
31. cout<<"Before sorting array elements are - "<<endl;
32. printArr(a, n);
33. insert(a, n);
34. cout<<"\nAfter sorting array elements are - "<<endl;
35. printArr(a, n);
37. **return** 0;
38. }

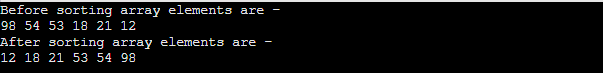
**Output:**



**Program:** Write a program to implement insertion sort in C# language.

1. **using** System;
2. **class** Insertion {
3. **static** **void** insert(**int**[] a) /\* function to sort an aay with insertion sort \*/
4. {
5. **int** i, j, temp;
6. **int** n = a.Length;
7. **for** (i = 1; i < n; i++) {
8. temp = a[i];
9. j = i - 1;
11. **while**(j>=0 && temp <= a[j])  /\* Move the elements greater than temp to one position ahead from their current position\*/
12. {
13. a[j+1] = a[j];
14. j = j-1;
15. }
16. a[j+1] = temp;
17. }
18. }
20. **static** **void** printArr(**int**[] a) /\* function to print the array \*/
21. {
22. **int** i;
23. **int** n = a.Length;
24. **for** (i = 0; i < n; i++)
25. Console.Write(a[i] + " ");
26. }
27. **static** **void** Main() {
28. **int**[] a = { 98, 54, 53, 18, 21, 12 };
29. Console.Write("Before sorting array elements are - \n");
30. printArr(a);
31. insert(a);
32. Console.Write("\nAfter sorting array elements are - \n");
33. printArr(a);
34. }
35. }

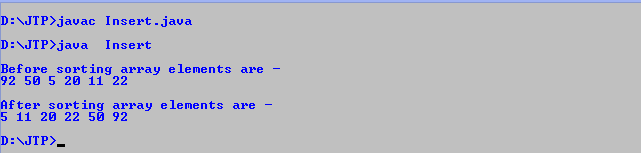
**Output:**



**Program:** Write a program to implement insertion sort in Java.

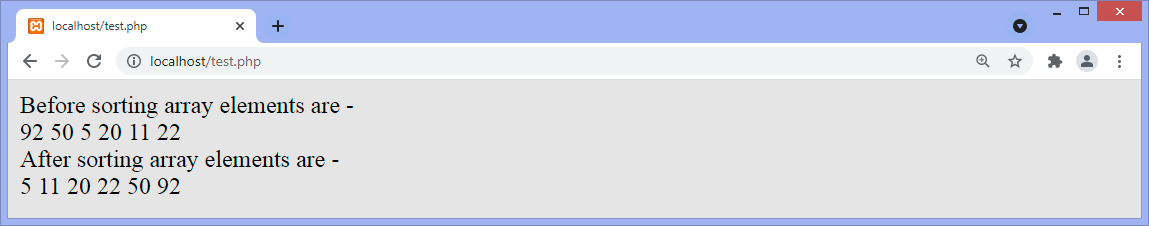
1. **public** **class** Insert
2. {
3. **void** insert(**int** a[]) /\* function to sort an aay with insertion sort \*/
4. {
5. **int** i, j, temp;
6. **int** n = a.length;
7. **for** (i = 1; i < n; i++) {
8. temp = a[i];
9. j = i - 1;
11. **while**(j>=0 && temp <= a[j])  /\* Move the elements greater than temp to one position ahead from their current position\*/
12. {
13. a[j+1] = a[j];
14. j = j-1;
15. }
16. a[j+1] = temp;
17. }
18. }
19. **void** printArr(**int** a[]) /\* function to print the array \*/
20. {
21. **int** i;
22. **int** n = a.length;
23. **for** (i = 0; i < n; i++)
24. System.out.print(a[i] + " ");
25. }
27. **public** **static** **void** main(String[] args) {
28. **int** a[] = { 92, 50, 5, 20, 11, 22 };
29. Insert i1 = **new** Insert();
30. System.out.println("\nBefore sorting array elements are - ");
31. i1.printArr(a);
32. i1.insert(a);
33. System.out.println("\n\nAfter sorting array elements are - ");
34. i1.printArr(a);
35. System.out.println();
36. }
37. }

**Output:**



**Program:** Write a program to implement insertion sort in PHP.



**Output:**

So, that's all about the article. Hope the article will be helpful and informative to you.

This article was not only limited to the algorithm. We have also discussed the algorithm's complexity, working, and implementation in different programming languages.

Merge Sort Algorithm

PlayNext

Unmute

Current TimeÂ 0:00

In this article, we will discuss the merge sort Algorithm. Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Now, let's see the algorithm of merge sort.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

1. MERGE\_SORT(arr, beg, end)
3. **if** beg < end
4. set mid = (beg + end)/2
5. MERGE\_SORT(arr, beg, mid)
6. MERGE\_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
10. END MERGE\_SORT

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

The implementation of the **MERGE** function is given as follows -

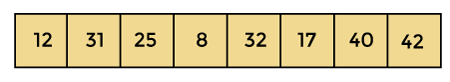
1. /\* Function to merge the subarrays of a[] \*/
2. **void** merge(**int** a[], **int** beg, **int** mid, **int** end)
3. {
4. **int** i, j, k;
5. **int** n1 = mid - beg + 1;
6. **int** n2 = end - mid;
8. **int** LeftArray[n1], RightArray[n2]; //temporary arrays
10. /\* copy data to temp arrays \*/
11. **for** (**int** i = 0; i < n1; i++)
12. LeftArray[i] = a[beg + i];
13. **for** (**int** j = 0; j < n2; j++)
14. RightArray[j] = a[mid + 1 + j];
16. i = 0, /\* initial index of first sub-array \*/
17. j = 0; /\* initial index of second sub-array \*/
18. k = beg;  /\* initial index of merged sub-array \*/
20. **while** (i < n1 && j < n2)
21. {
22. **if**(LeftArray[i] <= RightArray[j])
23. {
24. a[k] = LeftArray[i];
25. i++;
26. }
27. **else**
28. {
29. a[k] = RightArray[j];
30. j++;
31. }
32. k++;
33. }
34. **while** (i<n1)
35. {
36. a[k] = LeftArray[i];
37. i++;
38. k++;
39. }
41. **while** (j<n2)
42. {
43. a[k] = RightArray[j];
44. j++;
45. k++;
46. }
47. }

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

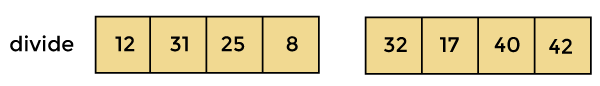
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

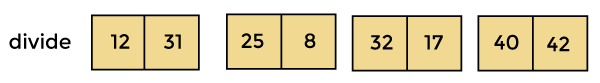


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

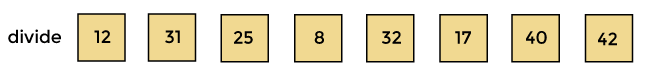
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



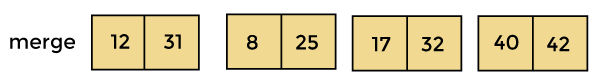
Now, again divide these arrays to get the atomic value that cannot be further divided.



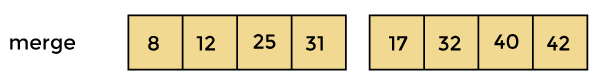
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

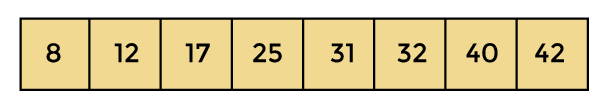
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n\*logn) |
| **Average Case** | O(n\*logn) |
| **Worst Case** | O(n\*logn) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n\*logn)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n\*logn)**.

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(n) |
| **Stable** | YES |

* The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

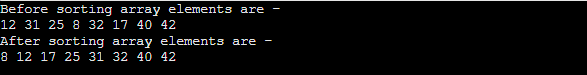
Implementation of merge sort

Now, let's see the programs of merge sort in different programming languages.

**Program:** Write a program to implement merge sort in C language.

1. #include <stdio.h>
3. /\* Function to merge the subarrays of a[] \*/
4. **void** merge(**int** a[], **int** beg, **int** mid, **int** end)
5. {
6. **int** i, j, k;
7. **int** n1 = mid - beg + 1;
8. **int** n2 = end - mid;
10. **int** LeftArray[n1], RightArray[n2]; //temporary arrays
12. /\* copy data to temp arrays \*/
13. **for** (**int** i = 0; i < n1; i++)
14. LeftArray[i] = a[beg + i];
15. **for** (**int** j = 0; j < n2; j++)
16. RightArray[j] = a[mid + 1 + j];
18. i = 0; /\* initial index of first sub-array \*/
19. j = 0; /\* initial index of second sub-array \*/
20. k = beg;  /\* initial index of merged sub-array \*/
22. **while** (i < n1 && j < n2)
23. {
24. **if**(LeftArray[i] <= RightArray[j])
25. {
26. a[k] = LeftArray[i];
27. i++;
28. }
29. **else**
30. {
31. a[k] = RightArray[j];
32. j++;
33. }
34. k++;
35. }
36. **while** (i<n1)
37. {
38. a[k] = LeftArray[i];
39. i++;
40. k++;
41. }
43. **while** (j<n2)
44. {
45. a[k] = RightArray[j];
46. j++;
47. k++;
48. }
49. }
51. **void** mergeSort(**int** a[], **int** beg, **int** end)
52. {
53. **if** (beg < end)
54. {
55. **int** mid = (beg + end) / 2;
56. mergeSort(a, beg, mid);
57. mergeSort(a, mid + 1, end);
58. merge(a, beg, mid, end);
59. }
60. }
62. /\* Function to print the array \*/
63. **void** printArray(**int** a[], **int** n)
64. {
65. **int** i;
66. **for** (i = 0; i < n; i++)
67. printf("%d ", a[i]);
68. printf("\n");
69. }
71. **int** main()
72. {
73. **int** a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
74. **int** n = **sizeof**(a) / **sizeof**(a[0]);
75. printf("Before sorting array elements are - \n");
76. printArray(a, n);
77. mergeSort(a, 0, n - 1);
78. printf("After sorting array elements are - \n");
79. printArray(a, n);
80. **return** 0;
81. }

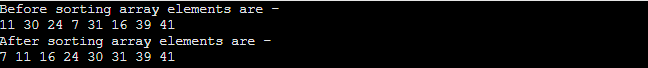
**Output:**



**Program:** Write a program to implement merge sort in C++ language.

1. #include <iostream>
3. **using** **namespace** std;
5. /\* Function to merge the subarrays of a[] \*/
6. **void** merge(**int** a[], **int** beg, **int** mid, **int** end)
7. {
8. **int** i, j, k;
9. **int** n1 = mid - beg + 1;
10. **int** n2 = end - mid;
12. **int** LeftArray[n1], RightArray[n2]; //temporary arrays
14. /\* copy data to temp arrays \*/
15. **for** (**int** i = 0; i < n1; i++)
16. LeftArray[i] = a[beg + i];
17. **for** (**int** j = 0; j < n2; j++)
18. RightArray[j] = a[mid + 1 + j];
20. i = 0; /\* initial index of first sub-array \*/
21. j = 0; /\* initial index of second sub-array \*/
22. k = beg;  /\* initial index of merged sub-array \*/
24. **while** (i < n1 && j < n2)
25. {
26. **if**(LeftArray[i] <= RightArray[j])
27. {
28. a[k] = LeftArray[i];
29. i++;
30. }
31. **else**
32. {
33. a[k] = RightArray[j];
34. j++;
35. }
36. k++;
37. }
38. **while** (i<n1)
39. {
40. a[k] = LeftArray[i];
41. i++;
42. k++;
43. }
45. **while** (j<n2)
46. {
47. a[k] = RightArray[j];
48. j++;
49. k++;
50. }
51. }
53. **void** mergeSort(**int** a[], **int** beg, **int** end)
54. {
55. **if** (beg < end)
56. {
57. **int** mid = (beg + end) / 2;
58. mergeSort(a, beg, mid);
59. mergeSort(a, mid + 1, end);
60. merge(a, beg, mid, end);
61. }
62. }
64. /\* Function to print the array \*/
65. **void** printArray(**int** a[], **int** n)
66. {
67. **int** i;
68. **for** (i = 0; i < n; i++)
69. cout<<a[i]<<" ";
70. }
72. **int** main()
73. {
74. **int** a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };
75. **int** n = **sizeof**(a) / **sizeof**(a[0]);
76. cout<<"Before sorting array elements are - \n";
77. printArray(a, n);
78. mergeSort(a, 0, n - 1);
79. cout<<"\nAfter sorting array elements are - \n";
80. printArray(a, n);
81. **return** 0;
82. }

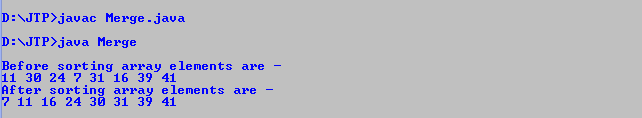
**Output:**



**Program:** Write a program to implement merge sort in Java.

1. **class** Merge {
3. /\* Function to merge the subarrays of a[] \*/
4. **void** merge(**int** a[], **int** beg, **int** mid, **int** end)
5. {
6. **int** i, j, k;
7. **int** n1 = mid - beg + 1;
8. **int** n2 = end - mid;
10. /\* temporary Arrays \*/
11. **int** LeftArray[] = **new** **int**[n1];
12. **int** RightArray[] = **new** **int**[n2];
14. /\* copy data to temp arrays \*/
15. **for** (i = 0; i < n1; i++)
16. LeftArray[i] = a[beg + i];
17. **for** (j = 0; j < n2; j++)
18. RightArray[j] = a[mid + 1 + j];
20. i = 0; /\* initial index of first sub-array \*/
21. j = 0; /\* initial index of second sub-array \*/
22. k = beg;  /\* initial index of merged sub-array \*/
24. **while** (i < n1 && j < n2)
25. {
26. **if**(LeftArray[i] <= RightArray[j])
27. {
28. a[k] = LeftArray[i];
29. i++;
30. }
31. **else**
32. {
33. a[k] = RightArray[j];
34. j++;
35. }
36. k++;
37. }
38. **while** (i<n1)
39. {
40. a[k] = LeftArray[i];
41. i++;
42. k++;
43. }
45. **while** (j<n2)
46. {
47. a[k] = RightArray[j];
48. j++;
49. k++;
50. }
51. }
53. **void** mergeSort(**int** a[], **int** beg, **int** end)
54. {
55. **if** (beg < end)
56. {
57. **int** mid = (beg + end) / 2;
58. mergeSort(a, beg, mid);
59. mergeSort(a, mid + 1, end);
60. merge(a, beg, mid, end);
61. }
62. }
64. /\* Function to print the array \*/
65. **void** printArray(**int** a[], **int** n)
66. {
67. **int** i;
68. **for** (i = 0; i < n; i++)
69. System.out.print(a[i] + " ");
70. }
72. **public** **static** **void** main(String args[])
73. {
74. **int** a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };
75. **int** n = a.length;
76. Merge m1 = **new** Merge();
77. System.out.println("\nBefore sorting array elements are - ");
78. m1.printArray(a, n);
79. m1.mergeSort(a, 0, n - 1);
80. System.out.println("\nAfter sorting array elements are - ");
81. m1.printArray(a, n);
82. System.out.println("");
83. }
85. }

**Output:**



**Program:** Write a program to implement merge sort in C#.

1. **using** System;
2. **class** Merge {
4. /\* Function to merge the subarrays of a[] \*/
5. **static** **void** merge(**int**[] a, **int** beg, **int** mid, **int** end)
6. {
7. **int** i, j, k;
8. **int** n1 = mid - beg + 1;
9. **int** n2 = end - mid;
11. //temporary arrays
12. **int**[] LeftArray = **new** **int** [n1];
13. **int**[] RightArray = **new** **int** [n2];
15. /\* copy data to temp arrays \*/
16. **for** (i = 0; i < n1; i++)
17. LeftArray[i] = a[beg + i];
18. **for** (j = 0; j < n2; j++)
19. RightArray[j] = a[mid + 1 + j];
21. i = 0; /\* initial index of first sub-array \*/
22. j = 0; /\* initial index of second sub-array \*/
23. k = beg;  /\* initial index of merged sub-array \*/
25. **while** (i < n1 && j < n2)
26. {
27. **if**(LeftArray[i] <= RightArray[j])
28. {
29. a[k] = LeftArray[i];
30. i++;
31. }
32. **else**
33. {
34. a[k] = RightArray[j];
35. j++;
36. }
37. k++;
38. }
39. **while** (i<n1)
40. {
41. a[k] = LeftArray[i];
42. i++;
43. k++;
44. }
46. **while** (j<n2)
47. {
48. a[k] = RightArray[j];
49. j++;
50. k++;
51. }
52. }
54. **static** **void** mergeSort(**int**[] a, **int** beg, **int** end)
55. {
56. **if** (beg < end)
57. {
58. **int** mid = (beg + end) / 2;
59. mergeSort(a, beg, mid);
60. mergeSort(a, mid + 1, end);
61. merge(a, beg, mid, end);
62. }
63. }
65. /\* Function to print the array \*/
66. **static** **void** printArray(**int**[] a, **int** n)
67. {
68. **int** i;
69. **for** (i = 0; i < n; i++)
70. Console.Write(a[i] + " ");
71. }
73. **static** **void** Main()
74. {
75. **int**[] a = { 10, 29, 23, 6, 30, 15, 38, 40 };
76. **int** n = a.Length;
77. Console.Write("Before sorting array elements are - ");
78. printArray(a, n);
79. mergeSort(a, 0, n - 1);
80. Console.Write("\nAfter sorting array elements are - ");
81. printArray(a, n);
82. }
84. }

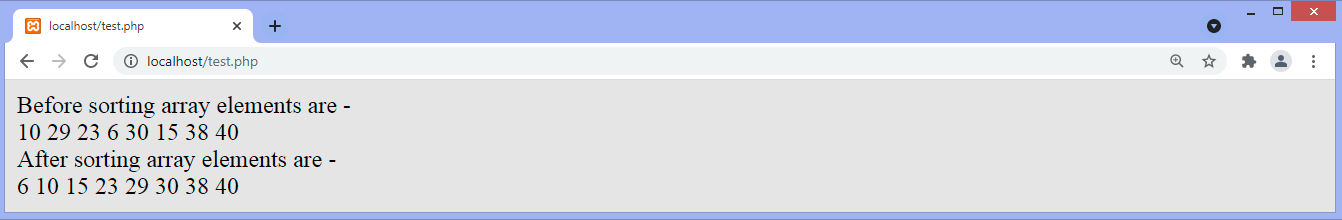
**Output:**

Merge sort

**Program:** Write a program to implement merge sort in PHP.

1. <?php
3. /\* Function to merge the subarrays of a[] \*/
4. function merge(&$a, $beg, $mid, $end)
5. {
6. $n1 = ($mid - $beg) + 1;
7. $n2 = $end - $mid;
9. /\* temporary Arrays \*/
10. $LeftArray = array($n1);
11. $RightArray = array($n2);
13. /\* copy data to temp arrays \*/
14. **for** ($i = 0; $i < $n1; $i++)
15. $LeftArray[$i] = $a[$beg + $i];
16. **for** ($j = 0; $j < $n2; $j++)
17. $RightArray[$j] = $a[$mid + 1 + $j];
19. $i = 0; /\* initial index of first sub-array \*/
20. $j = 0; /\* initial index of second sub-array \*/
21. $k = $beg;  /\* initial index of merged sub-array \*/
23. **while** ($i<$n1 && $j<$n2)
24. {
25. **if**($LeftArray[$i] <= $RightArray[$j])
26. {
27. $a[$k] = $LeftArray[$i];
28. $i++;
29. }
30. **else**
31. {
32. $a[$k] = $RightArray[$j];
33. $j++;
34. }
35. $k++;
36. }
37. **while** ($i<$n1)
38. {
39. $a[$k] = $LeftArray[$i];
40. $i++;
41. $k++;
42. }
44. **while** ($j<$n2)
45. {
46. $a[$k] = $RightArray[$j];
47. $j++;
48. $k++;
49. }
50. }
52. function mergeSort(&$a, $beg, $end)
53. {
54. **if** ($beg < $end)
55. {
56. $mid = (**int**)(($beg + $end) / 2);
57. mergeSort($a, $beg, $mid);
58. mergeSort($a, $mid + 1, $end);
59. merge($a, $beg, $mid, $end);
60. }
61. }
63. /\* Function to print array elements \*/
64. function printArray($a, $n)
65. {
66. **for**($i = 0; $i < $n; $i++)
67. {
68. print\_r($a[$i]);
69. echo " ";
70. }
71. }
73. $a = array( 10, 29, 23, 6, 30, 15, 38, 40 );
74. $n = count($a);
75. echo "Before sorting array elements are - <br>";
76. printArray($a, $n);
77. mergeSort($a, 0, $n - 1);
78. echo "<br> After sorting array elements are - <br>";
79. printArray($a, $n);
80. ?>

**Output:**



So, that's all about the article. Hope the article will be helpful and informative to you.

This article was not only limited to the algorithm. We have also discussed the Merge sort complexity, working, and implementation in different programming languages.

Quick Sort Algorithm

In this article, we will discuss the Quicksort Algorithm. The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

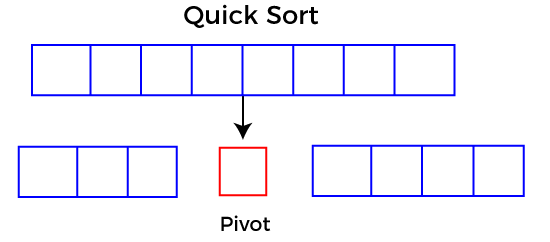
**Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively, sort two subarrays with Quicksort.

**Combine:** Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

* Pivot can be random, i.e. select the random pivot from the given array.
* Pivot can either be the rightmost element of the leftmost element of the given array.
* Select median as the pivot element.

Algorithm

**Algorithm:**

1. QUICKSORT (array A, start, end)
2. {
3. 1 **if** (start < end)
4. 2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }

**Partition Algorithm:**

The partition algorithm rearranges the sub-arrays in a place.

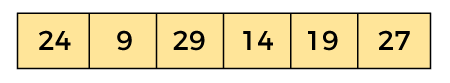
1. PARTITION (array A, start, end)
2. {
3. 1 pivot ? A[end]
4. 2 i ? start-1
5. 3 **for** j ? start to end -1 {
6. 4 **do** **if** (A[j] < pivot) {
7. 5 then i ? i + 1
8. 6 swap A[i] with A[j]
9. 7  }}
10. 8 swap A[i+1] with A[end]
11. 9 **return** i+1
12. }

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

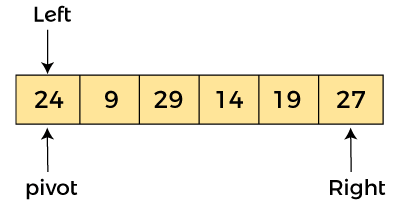
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

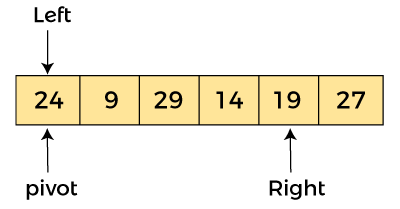


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

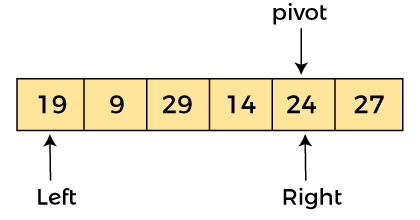


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



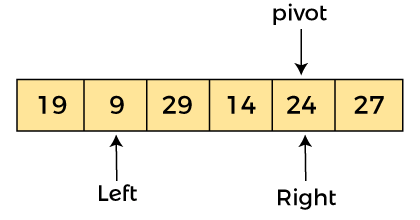
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

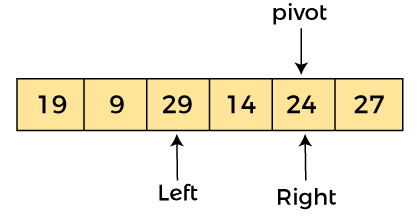


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

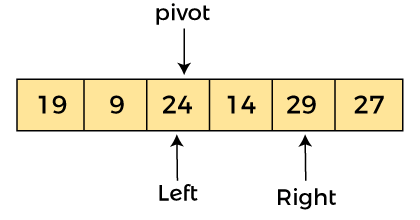
As a[pivot] > a[left], so algorithm moves one position to right as -



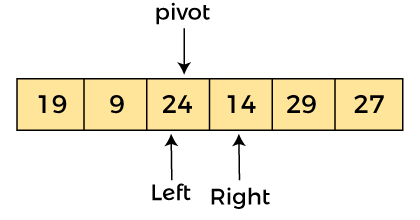
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



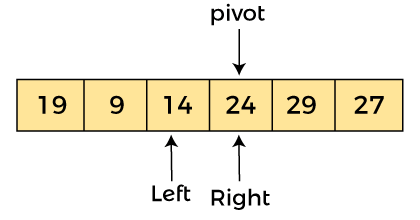
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



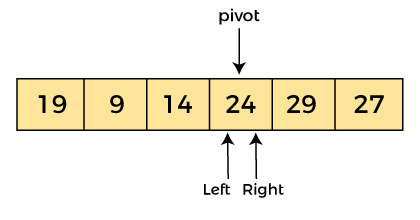
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



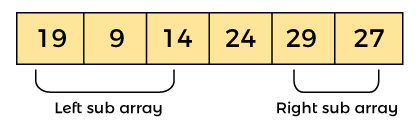
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



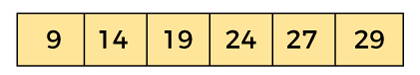
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n\*logn) |
| **Average Case** | O(n\*logn) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n\*logn)**.
* **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n2)**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(n\*logn) |
| **Stable** | NO |

* The space complexity of quicksort is O(n\*logn).

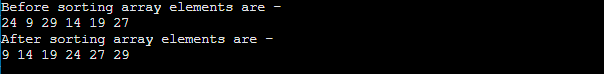
Implementation of quicksort

Now, let's see the programs of quicksort in different programming languages.

**Program:** Write a program to implement quicksort in C language.

1. #include <stdio.h>
2. /\* function that consider last element as pivot,
3. place the pivot at its exact position, and place
4. smaller elements to left of pivot and greater
5. elements to right of pivot.  \*/
6. **int** partition (**int** a[], **int** start, **int** end)
7. {
8. **int** pivot = a[end]; // pivot element
9. **int** i = (start - 1);
11. **for** (**int** j = start; j <= end - 1; j++)
12. {
13. // If current element is smaller than the pivot
14. **if** (a[j] < pivot)
15. {
16. i++; // increment index of smaller element
17. **int** t = a[i];
18. a[i] = a[j];
19. a[j] = t;
20. }
21. }
22. **int** t = a[i+1];
23. a[i+1] = a[end];
24. a[end] = t;
25. **return** (i + 1);
26. }
28. /\* function to implement quick sort \*/
29. **void** quick(**int** a[], **int** start, **int** end) /\* a[] = array to be sorted, start = Starting index, end = Ending index \*/
30. {
31. **if** (start < end)
32. {
33. **int** p = partition(a, start, end); //p is the partitioning index
34. quick(a, start, p - 1);
35. quick(a, p + 1, end);
36. }
37. }
39. /\* function to print an array \*/
40. **void** printArr(**int** a[], **int** n)
41. {
42. **int** i;
43. **for** (i = 0; i < n; i++)
44. printf("%d ", a[i]);
45. }
46. **int** main()
47. {
48. **int** a[] = { 24, 9, 29, 14, 19, 27 };
49. **int** n = **sizeof**(a) / **sizeof**(a[0]);
50. printf("Before sorting array elements are - \n");
51. printArr(a, n);
52. quick(a, 0, n - 1);
53. printf("\nAfter sorting array elements are - \n");
54. printArr(a, n);
56. **return** 0;
57. }

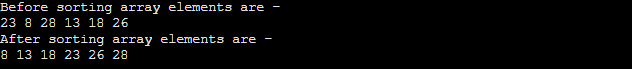
**Output:**



**Program:** Write a program to implement quick sort in C++ language.

1. #include <iostream>
3. **using** **namespace** std;
5. /\* function that consider last element as pivot,
6. place the pivot at its exact position, and place
7. smaller elements to left of pivot and greater
8. elements to right of pivot.  \*/
9. **int** partition (**int** a[], **int** start, **int** end)
10. {
11. **int** pivot = a[end]; // pivot element
12. **int** i = (start - 1);
14. **for** (**int** j = start; j <= end - 1; j++)
15. {
16. // If current element is smaller than the pivot
17. **if** (a[j] < pivot)
18. {
19. i++; // increment index of smaller element
20. **int** t = a[i];
21. a[i] = a[j];
22. a[j] = t;
23. }
24. }
25. **int** t = a[i+1];
26. a[i+1] = a[end];
27. a[end] = t;
28. **return** (i + 1);
29. }
31. /\* function to implement quick sort \*/
32. **void** quick(**int** a[], **int** start, **int** end) /\* a[] = array to be sorted, start = Starting index, end = Ending index \*/
33. {
34. **if** (start < end)
35. {
36. **int** p = partition(a, start, end);  //p is the partitioning index
37. quick(a, start, p - 1);
38. quick(a, p + 1, end);
39. }
40. }
42. /\* function to print an array \*/
43. **void** printArr(**int** a[], **int** n)
44. {
45. **int** i;
46. **for** (i = 0; i < n; i++)
47. cout<<a[i]<< " ";
48. }
49. **int** main()
50. {
51. **int** a[] = { 23, 8, 28, 13, 18, 26 };
52. **int** n = **sizeof**(a) / **sizeof**(a[0]);
53. cout<<"Before sorting array elements are - \n";
54. printArr(a, n);
55. quick(a, 0, n - 1);
56. cout<<"\nAfter sorting array elements are - \n";
57. printArr(a, n);
59. **return** 0;
60. }

**Output:**



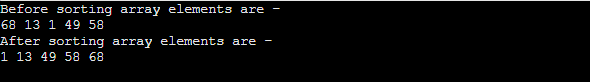
**Program:** Write a program to implement quicksort in python.

1. #function that consider last element as pivot,
2. #place the pivot at its exact position, and place
3. #smaller elements to left of pivot and greater
4. #elements to right of pivot.
6. def partition (a, start, end):
7. i = (start - 1)
8. pivot = a[end] # pivot element
10. **for** j in range(start, end):
11. # If current element is smaller than or equal to the pivot
12. **if** (a[j] <= pivot):
13. i = i + 1
14. a[i], a[j] = a[j], a[i]
16. a[i+1], a[end] = a[end], a[i+1]
18. **return** (i + 1)
20. # function to implement quick sort
21. def quick(a, start, end): # a[] = array to be sorted, start = Starting index, end = Ending index
22. **if** (start < end):
23. p = partition(a, start, end) # p is partitioning index
24. quick(a, start, p - 1)
25. quick(a, p + 1, end)

28. def printArr(a): # function to print the array
29. **for** i in range(len(a)):
30. print (a[i], end = " ")

33. a = [68, 13, 1, 49, 58]
34. print("Before sorting array elements are - ")
35. printArr(a)
36. quick(a, 0, len(a)-1)
37. print("\nAfter sorting array elements are - ")
38. printArr(a)

**Output:**

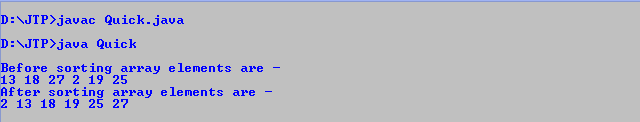


**Program:** Write a program to implement quicksort in Java.

1. **public** **class** Quick
2. {
3. /\* function that consider last element as pivot,
4. place the pivot at its exact position, and place
5. smaller elements to left of pivot and greater
6. elements to right of pivot.  \*/
7. **int** partition (**int** a[], **int** start, **int** end)
8. {
9. **int** pivot = a[end]; // pivot element
10. **int** i = (start - 1);
12. **for** (**int** j = start; j <= end - 1; j++)
13. {
14. // If current element is smaller than the pivot
15. **if** (a[j] < pivot)
16. {
17. i++; // increment index of smaller element
18. **int** t = a[i];
19. a[i] = a[j];
20. a[j] = t;
21. }
22. }
23. **int** t = a[i+1];
24. a[i+1] = a[end];
25. a[end] = t;
26. **return** (i + 1);
27. }
29. /\* function to implement quick sort \*/
30. **void** quick(**int** a[], **int** start, **int** end) /\* a[] = array to be sorted, start = Starting index, end = Ending index \*/
31. {
32. **if** (start < end)
33. {
34. **int** p = partition(a, start, end);  //p is partitioning index
35. quick(a, start, p - 1);
36. quick(a, p + 1, end);
37. }
38. }
40. /\* function to print an array \*/
41. **void** printArr(**int** a[], **int** n)
42. {
43. **int** i;
44. **for** (i = 0; i < n; i++)
45. System.out.print(a[i] + " ");
46. }
47. **public** **static** **void** main(String[] args) {
48. **int** a[] = { 13, 18, 27, 2, 19, 25 };
49. **int** n = a.length;
50. System.out.println("\nBefore sorting array elements are - ");
51. Quick q1 = **new** Quick();
52. q1.printArr(a, n);
53. q1.quick(a, 0, n - 1);
54. System.out.println("\nAfter sorting array elements are - ");
55. q1.printArr(a, n);
56. System.out.println();
57. }
58. }

**Output**

After the execution of above code, the output will be -

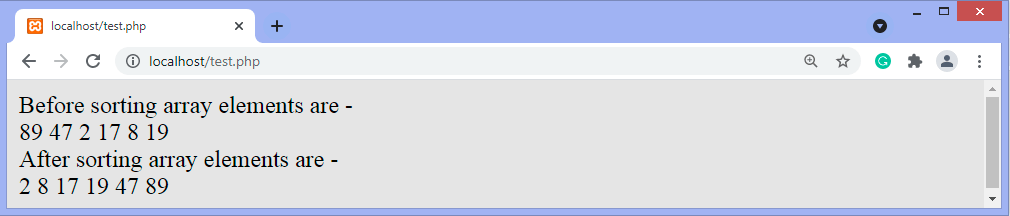


**Program:** Write a program to implement quick sort in php.

1. **<?php**
2. /\* function that consider last element as pivot,
3. place the pivot at its exact position, and place
4. smaller elements to left of pivot and greater
5. elements to right of pivot.  \*/
6. function partition (&$a, $start, $end)
7. {
8. $pivot = $a[$end]; // pivot element
9. $i = ($start - 1);
11. for ($j = $start; $j **<**= $end - 1; $j++)
12. {
13. // If current element is smaller than the pivot
14. if ($a[$j] **<** $pivot)
15. {
16. $i++; // increment index of smaller element
17. $t = $a[$i];
18. $a[$i] = $a[$j];
19. $a[$j] = $t;
20. }
21. }
22. $t = $a[$i+1];
23. $a[$i+1] = $a[$end];
24. $a[$end] = $t;
25. return ($i + 1);
26. }
28. /\* function to implement quick sort \*/
29. function quick(&$a, $start, $end) /\* a[] = array to be sorted, start = Starting index, end = Ending index \*/
30. {
31. if ($start **<** $end)
32. {
33. $p = partition($a, $start, $end); //p is partitioning index
34. quick($a, $start, $p - 1);
35. quick($a, $p + 1, $end);
36. }
37. }
39. function printArray($a, $n)
40. {
41. for($i = 0; $i **<** $n; $i++)
42. {
43. print\_r($a[$i]);
44. echo " ";
45. }
46. }
47. $a = array( 89, 47, 2, 17, 8, 19 );
48. $n = count($a);
49. echo "Before sorting array elements are - **<br>**";
50. printArray($a, $n);
51. quick($a, 0, $n - 1);
52. echo "**<br>** After sorting array elements are - **<br>**";
53. printArray($a, $n);
55. **?>**

**Output**

After the execution of above code, the output will be -



So, that's all about the article. Hope the article will be helpful and informative to you.

This article was not only limited to the algorithm. Along with the algorithm, we have also discussed the quick sort complexity, working, and implementation in different programming languages.

Radix Sort Algorithm

In this article, we will discuss the Radix sort Algorithm. Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

Now, let's see the algorithm of Radix sort.

Algorithm

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. **for** i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digits at
7. the ith place

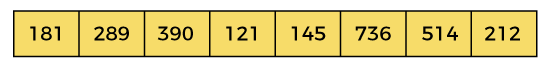
Working of Radix sort Algorithm

Now, let's see the working of Radix sort Algorithm.

The steps used in the sorting of radix sort are listed as follows -

* First, we have to find the largest element (suppose **max**) from the given array. Suppose **'x'** be the number of digits in **max**. The **'x'** is calculated because we need to go through the significant places of all elements.
* After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

Now let's see the working of radix sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using radix sort. It will make the explanation clearer and easier.

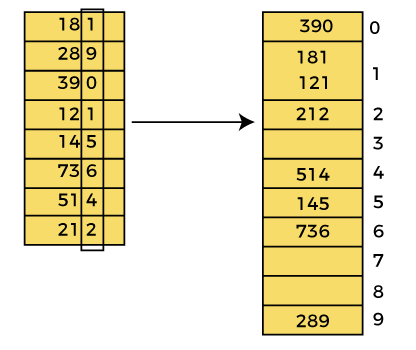


In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

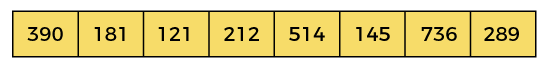
Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.

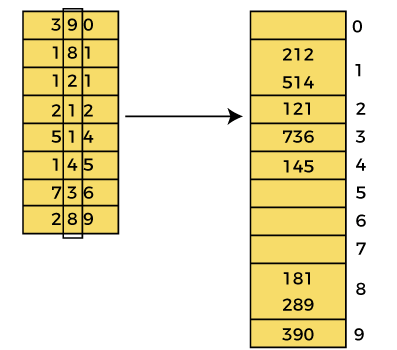


After the first pass, the array elements are -

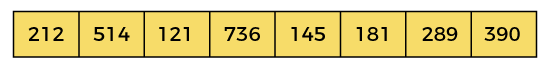


Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).

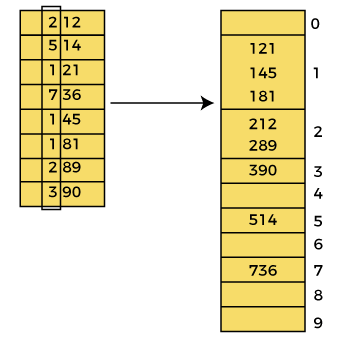


After the second pass, the array elements are -

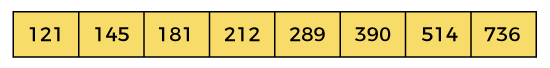


Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



After the third pass, the array elements are -



Now, the array is sorted in ascending order.

Radix sort complexity

Now, let's see the time complexity of Radix sort in best case, average case, and worst case. We will also see the space complexity of Radix sort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | Ω(n+k) |
| **Average Case** | θ(nk) |
| **Worst Case** | O(nk) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is **Ω(n+k)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is **θ(nk)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is **O(nk)**.

Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(n + k) |
| **Stable** | YES |

* The space complexity of Radix sort is O(n + k).

Implementation of Radix sort

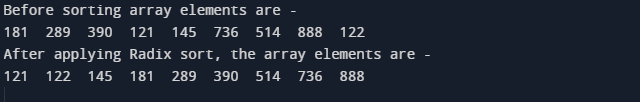
Now, let's see the programs of Radix sort in different programming languages.

**Program:** Write a program to implement Radix sort in C language.

1. #include <stdio.h>
3. **int** getMax(**int** a[], **int** n) {
4. **int** max = a[0];
5. **for**(**int** i = 1; i<n; i++) {
6. **if**(a[i] > max)
7. max = a[i];
8. }
9. **return** max; //maximum element from the array
10. }
12. **void** countingSort(**int** a[], **int** n, **int** place) // function to implement counting sort
13. {
14. **int** output[n + 1];
15. **int** count[10] = {0};
17. // Calculate count of elements
18. **for** (**int** i = 0; i < n; i++)
19. count[(a[i] / place) % 10]++;
21. // Calculate cumulative frequency
22. **for** (**int** i = 1; i < 10; i++)
23. count[i] += count[i - 1];
25. // Place the elements in sorted order
26. **for** (**int** i = n - 1; i >= 0; i--) {
27. output[count[(a[i] / place) % 10] - 1] = a[i];
28. count[(a[i] / place) % 10]--;
29. }
31. **for** (**int** i = 0; i < n; i++)
32. a[i] = output[i];
33. }
35. // function to implement radix sort
36. **void** radixsort(**int** a[], **int** n) {
38. // get maximum element from array
39. **int** max = getMax(a, n);
41. // Apply counting sort to sort elements based on place value
42. **for** (**int** place = 1; max / place > 0; place \*= 10)
43. countingSort(a, n, place);
44. }
46. // function to print array elements
47. **void** printArray(**int** a[], **int** n) {
48. **for** (**int** i = 0; i < n; ++i) {
49. printf("%d  ", a[i]);
50. }
51. printf("\n");
52. }
54. **int** main() {
55. **int** a[] = {181, 289, 390, 121, 145, 736, 514, 888, 122};
56. **int** n = **sizeof**(a) / **sizeof**(a[0]);
57. printf("Before sorting array elements are - \n");
58. printArray(a,n);
59. radixsort(a, n);
60. printf("After applying Radix sort, the array elements are - \n");
61. printArray(a, n);
62. }

**Output:**

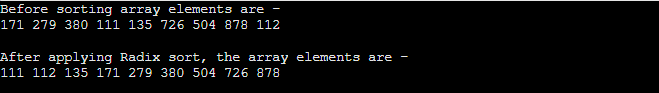
After the execution of the above code, the output will be -



**Program:** Write a program to implement Radix sort in C++.

1. #include <iostream>
3. **using** **namespace** std;
5. **int** getMax(**int** a[], **int** n) {
6. **int** max = a[0];
7. **for**(**int** i = 1; i<n; i++) {
8. **if**(a[i] > max)
9. max = a[i];
10. }
11. **return** max; //maximum element from the array
12. }
14. **void** countingSort(**int** a[], **int** n, **int** place) // function to implement counting sort
15. {
16. **int** output[n + 1];
17. **int** count[10] = {0};
19. // Calculate count of elements
20. **for** (**int** i = 0; i < n; i++)
21. count[(a[i] / place) % 10]++;
23. // Calculate cumulative frequency
24. **for** (**int** i = 1; i < 10; i++)
25. count[i] += count[i - 1];
27. // Place the elements in sorted order
28. **for** (**int** i = n - 1; i >= 0; i--) {
29. output[count[(a[i] / place) % 10] - 1] = a[i];
30. count[(a[i] / place) % 10]--;
31. }
33. **for** (**int** i = 0; i < n; i++)
34. a[i] = output[i];
35. }
37. // function to implement radix sort
38. **void** radixsort(**int** a[], **int** n) {
40. // get maximum element from array
41. **int** max = getMax(a, n);
43. // Apply counting sort to sort elements based on place value
44. **for** (**int** place = 1; max / place > 0; place \*= 10)
45. countingSort(a, n, place);
46. }
48. // function to print array elements
49. **void** printArray(**int** a[], **int** n) {
50. **for** (**int** i = 0; i < n; ++i)
51. cout<<a[i]<<" ";
52. }
54. **int** main() {
55. **int** a[] = {171, 279, 380, 111, 135, 726, 504, 878, 112};
56. **int** n = **sizeof**(a) / **sizeof**(a[0]);
57. cout<<"Before sorting array elements are - \n";
58. printArray(a,n);
59. radixsort(a, n);
60. cout<<"\n\nAfter applying Radix sort, the array elements are - \n";
61. printArray(a, n);
62. **return** 0;
63. }

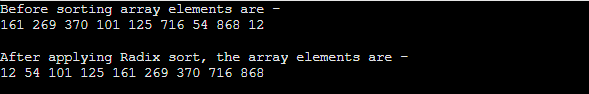
**Output:**



**Program:** Write a program to implement Radix sort in C#.

1. **using** System;
2. **class** RadixSort {
4. **static** **int** getMax(**int**[] a, **int** n) {
5. **int** max = a[0];
6. **for**(**int** i = 1; i<n; i++) {
7. **if**(a[i] > max)
8. max = a[i];
9. }
10. **return** max; //maximum element from the array
11. }
13. **static** **void** countingSort(**int**[] a, **int** n, **int** place) // function to implement counting sort
14. {
15. **int**[] output = **new** **int**[n+1];
16. **int**[] count = **new** **int**[10];
18. // Calculate count of elements
19. **for** (**int** i = 0; i < n; i++)
20. count[(a[i] / place) % 10]++;
22. // Calculate cumulative frequency
23. **for** (**int** i = 1; i < 10; i++)
24. count[i] += count[i - 1];
26. // Place the elements in sorted order
27. **for** (**int** i = n - 1; i >= 0; i--) {
28. output[count[(a[i] / place) % 10] - 1] = a[i];
29. count[(a[i] / place) % 10]--;
30. }
32. **for** (**int** i = 0; i < n; i++)
33. a[i] = output[i];
34. }
36. // function to implement radix sort
37. **static** **void** radixsort(**int**[] a, **int** n) {
39. // get maximum element from array
40. **int** max = getMax(a, n);
42. // Apply counting sort to sort elements based on place value
43. **for** (**int** place = 1; max / place > 0; place \*= 10)
44. countingSort(a, n, place);
45. }
47. // function to print array elements
48. **static** **void** printArray(**int**[] a, **int** n) {
49. **for** (**int** i = 0; i < n; ++i)
50. Console.Write(a[i] + " ");
51. }
53. **static** **void** Main() {
54. **int**[] a = {161, 269, 370, 101, 125, 716, 54, 868, 12};
55. **int** n = a.Length;
56. Console.Write("Before sorting array elements are - \n");
57. printArray(a,n);
58. radixsort(a, n);
59. Console.Write("\n\nAfter applying Radix sort, the array elements are - \n");
60. printArray(a, n);
61. }
62. }

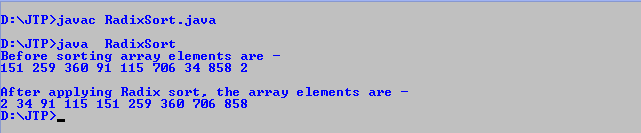
**Output:**



**Program:** Write a program to implement Radix sort in Java.

1. **class** RadixSort {
3. **int** getMax(**int** a[], **int** n) {
4. **int** max = a[0];
5. **for**(**int** i = 1; i<n; i++) {
6. **if**(a[i] > max)
7. max = a[i];
8. }
9. **return** max; //maximum element from the array
10. }
12. **void** countingSort(**int** a[], **int** n, **int** place) // function to implement counting
14. sort
15. {
16. **int**[] output = **new** **int**[n+1];
17. **int**[] count = **new** **int**[10];
19. // Calculate count of elements
20. **for** (**int** i = 0; i < n; i++)
21. count[(a[i] / place) % 10]++;
23. // Calculate cumulative frequency
24. **for** (**int** i = 1; i < 10; i++)
25. count[i] += count[i - 1];
27. // Place the elements in sorted order
28. **for** (**int** i = n - 1; i >= 0; i--) {
29. output[count[(a[i] / place) % 10] - 1] = a[i];
30. count[(a[i] / place) % 10]--;
31. }
33. **for** (**int** i = 0; i < n; i++)
34. a[i] = output[i];
35. }
37. // function to implement radix sort
38. **void** radixsort(**int** a[], **int** n) {
40. // get maximum element from array
41. **int** max = getMax(a, n);
43. // Apply counting sort to sort elements based on place value
44. **for** (**int** place = 1; max / place > 0; place \*= 10)
45. countingSort(a, n, place);
46. }
48. // function to print array elements
49. **void** printArray(**int** a[], **int** n) {
50. **for** (**int** i = 0; i < n; ++i)
51. System.out.print(a[i] + " ");
52. }
54. **public** **static** **void** main(String args[]) {
55. **int** a[] = {151, 259, 360, 91, 115, 706, 34, 858, 2};
56. **int** n = a.length;
57. RadixSort r1 = **new** RadixSort();
58. System.out.print("Before sorting array elements are - \n");
59. r1.printArray(a,n);
60. r1.radixsort(a, n);
61. System.out.print("\n\nAfter applying Radix sort, the array elements are -
63. \n");
64. r1.printArray(a, n);
65. }
66. }

**Output:**



So, that's all about the article. Hope the article will be helpful and informative to you.

Selection Sort Algorithm

In this article, we will discuss the Selection sort Algorithm. The working procedure of selection sort is also simple. This article will be very helpful and interesting to students as they might face selection sort as a question in their examinations. So, it is important to discuss the topic.

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is **O(n2)**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

* A small array is to be sorted
* Swapping cost doesn't matter
* It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

Algorithm

1. SELECTION SORT(arr, n)
3. Step 1: Repeat Steps 2 **and** 3 **for** i = 0 to n-1
4. Step 2: CALL SMALLEST(arr, i, n, pos)
5. Step 3: SWAP arr[i] with arr[pos]
6. [END OF LOOP]
7. Step 4: EXIT
9. SMALLEST (arr, i, n, pos)
10. Step 1: [INITIALIZE] SET SMALL = arr[i]
11. Step 2: [INITIALIZE] SET pos = i
12. Step 3: Repeat **for** j = i+1 to n
13. **if** (SMALL > arr[j])
14. SET SMALL = arr[j]
15. SET pos = j
16. [END OF **if**]
17. [END OF LOOP]
18. Step 4: RETURN pos

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

selection Sort Algorithm

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

selection Sort Algorithm

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

selection Sort Algorithm

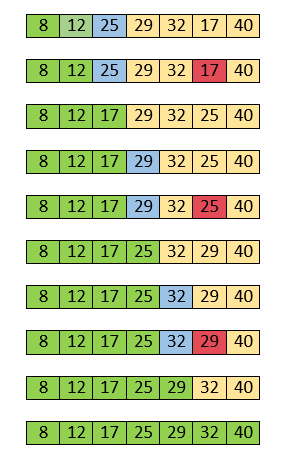
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

selection Sort Algorithm

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

selection Sort Algorithm

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n2) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n2)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n2)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n2)**.

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | YES |

* The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

Implementation of selection sort

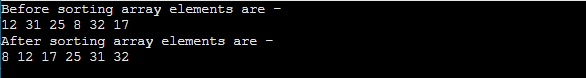
Now, let's see the programs of selection sort in different programming languages.

**Program:** Write a program to implement selection sort in C language.

1. #include <stdio.h>
3. **void** selection(**int** arr[], **int** n)
4. {
5. **int** i, j, small;
7. **for** (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
8. {
9. small = i; //minimum element in unsorted array
11. **for** (j = i+1; j < n; j++)
12. **if** (arr[j] < arr[small])
13. small = j;
14. // Swap the minimum element with the first element
15. **int** temp = arr[small];
16. arr[small] = arr[i];
17. arr[i] = temp;
18. }
19. }
21. **void** printArr(**int** a[], **int** n) /\* function to print the array \*/
22. {
23. **int** i;
24. **for** (i = 0; i < n; i++)
25. printf("%d ", a[i]);
26. }
28. **int** main()
29. {
30. **int** a[] = { 12, 31, 25, 8, 32, 17 };
31. **int** n = **sizeof**(a) / **sizeof**(a[0]);
32. printf("Before sorting array elements are - \n");
33. printArr(a, n);
34. selection(a, n);
35. printf("\nAfter sorting array elements are - \n");
36. printArr(a, n);
37. **return** 0;
38. }

**Output:**

After the execution of above code, the output will be -

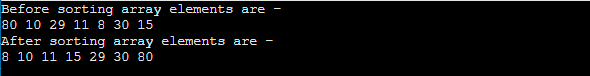


**Program:** Write a program to implement selection sort in C++ language.

1. #include <iostream>
3. **using** **namespace** std;
5. **void** selection(**int** arr[], **int** n)
6. {
7. **int** i, j, small;
9. **for** (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
10. {
11. small = i; //minimum element in unsorted array
13. **for** (j = i+1; j < n; j++)
14. **if** (arr[j] < arr[small])
15. small = j;
16. // Swap the minimum element with the first element
17. **int** temp = arr[small];
18. arr[small] = arr[i];
19. arr[i] = temp;
20. }
21. }
23. **void** printArr(**int** a[], **int** n) /\* function to print the array \*/
24. {
25. **int** i;
26. **for** (i = 0; i < n; i++)
27. cout<< a[i] <<" ";
28. }
30. **int** main()
31. {
32. **int** a[] = { 80, 10, 29, 11, 8, 30, 15 };
33. **int** n = **sizeof**(a) / **sizeof**(a[0]);
34. cout<< "Before sorting array elements are - "<<endl;
35. printArr(a, n);
36. selection(a, n);
37. cout<< "\nAfter sorting array elements are - "<<endl;
38. printArr(a, n);
40. **return** 0;
41. }

**Output:**

After the execution of above code, the output will be -



**Program:** Write a program to implement selection sort in C# language.

1. **using** System;
2. **class** Selection {
3. **static** **void** selection(**int**[] arr)
4. {
5. **int** i, j, small;
6. **int** n = arr.Length;
7. **for** (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
8. {
9. small = i; //minimum element in unsorted array
11. **for** (j = i+1; j < n; j++)
12. **if** (arr[j] < arr[small])
13. small = j;
14. // Swap the minimum element with the first element
15. **int** temp = arr[small];
16. arr[small] = arr[i];
17. arr[i] = temp;
18. }
19. }
21. **static** **void** printArr(**int**[] a) /\* function to print the array \*/
22. {
23. **int** i;
24. **int** n = a.Length;
25. **for** (i = 0; i < n; i++)
26. Console.Write(a[i] + " ");
27. }
29. **static** **void** Main() {
30. **int**[] a = { 85, 50, 29, 18, 7, 30, 3};
31. Console.Write("Before sorting array elements are - ");
32. printArr(a);
33. selection(a);
34. Console.Write("\nAfter sorting array elements are - ");
35. printArr(a);
36. }
37. }

**Output:**

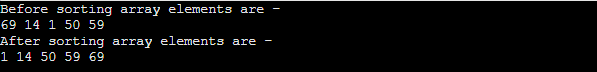
selection Sort Algorithm

**Program:** Write a program to implement selection sort in python.

1. def selection(a): # Function to implement selection sort
2. **for** i in range(len(a)): # Traverse through all array elements
3. small = i # minimum element in unsorted array
4. **for** j in range(i+1, len(a)):
5. **if** a[small] > a[j]:
6. small = j
7. # Swap the found minimum element with
8. # the first element
9. a[i], a[small] = a[small], a[i]
11. def printArr(a): # function to print the array
13. **for** i in range(len(a)):
14. print (a[i], end = " ")


18. a = [69, 14, 1, 50, 59]
19. print("Before sorting array elements are - ")
20. printArr(a)
21. selection(a)
22. print("\nAfter sorting array elements are - ")
23. selection(a)
24. printArr(a)

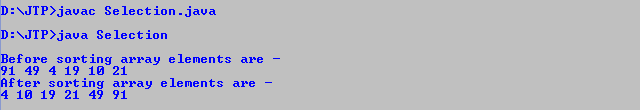
**Output:**



**Program:** Write a program to implement selection sort in Java.

1. **public** **class** Selection
2. {
3. **void** selection(**int** a[]) /\* function to sort an array with selection sort \*/
4. {
5. **int** i, j, small;
6. **int** n = a.length;
7. **for** (i = 0; i < n-1; i++)
8. {
9. small = i; //minimum element in unsorted array
11. **for** (j = i+1; j < n; j++)
12. **if** (a[j] < a[small])
13. small = j;
14. // Swap the minimum element with the first element
15. **int** temp = a[small];
16. a[small] = a[i];
17. a[i] = temp;
18. }
20. }
21. **void** printArr(**int** a[]) /\* function to print the array \*/
22. {
23. **int** i;
24. **int** n = a.length;
25. **for** (i = 0; i < n; i++)
26. System.out.print(a[i] + " ");
27. }
29. **public** **static** **void** main(String[] args) {
30. **int** a[] = { 91, 49, 4, 19, 10, 21 };
31. Selection i1 = **new** Selection();
32. System.out.println("\nBefore sorting array elements are - ");
33. i1.printArr(a);
34. i1.selection(a);
35. System.out.println("\nAfter sorting array elements are - ");
36. i1.printArr(a);
37. System.out.println();
38. }
39. }

**Output:**

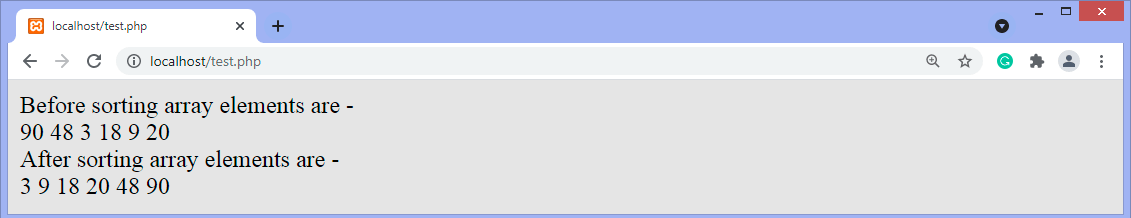


**Program:** Write a program to implement selection sort in PHP.

1. **<?php**
2. function selection(&$a, $n) /\* function to sort an array with selection sort \*/
3. {
4. for ($i = 0; $i **<** $n; $i++)
5. {
6. $small = $i; //minimum element in unsorted array
8. for ($j = $i+1; $j **<** $n; $j++)
9. if ($a[$j] **<** $a[$small])
10. $small = $j;
11. // Swap the minimum element with the first element
12. $temp = $a[$small];
13. $a[$small] = $a[$i];
14. $a[$i] = $temp;
15. }
17. }
18. function printArray($a, $n)
19. {
20. for($i = 0; $i **<** $n; $i++)
21. {
22. print\_r($a[$i]);
23. echo " ";
24. }
25. }
26. $a = array( 90, 48, 3, 18, 9, 20 );
27. $n = count($a);
28. echo "Before sorting array elements are - **<br>**";
29. printArray($a, $n);
30. selection($a, $n);
31. echo "**<br>** After sorting array elements are - **<br>**";
32. printArray($a, $n);
33. **?>**

**Output:**

After the execution of above code, the output will be -



So, that's all about the article. Hope the article will be helpful and informative to you.

This article was not only limited to the algorithm. We have also discussed the Selection sort complexity, working, and implementation in different programming languages.

Shell Sort Algorithm

In this article, we will discuss the shell sort algorithm. Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

It is a sorting algorithm that is an extended version of insertion sort. Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.

In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.

This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as **interval.** This interval can be calculated by using the **Knuth's** formula given below -

1. hh = h \* 3 + 1
2. where, 'h' is the interval having initial value 1.

Now, let's see the algorithm of shell sort.

Algorithm

The simple steps of achieving the shell sort are listed as follows -

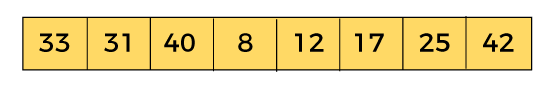
1. ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
2. for (interval = n/2; interval **>** 0; interval /= 2)
3. for ( i = interval; i **<** **n**; i += 1)
4. temp = a[i];
5. for (j = i; j **>**= interval && a[j - interval] **>** temp; j -= interval)
6. a[j] = a[j - interval];
7. a[j] = temp;
8. End ShellSort

Working of Shell sort Algorithm

Now, let's see the working of the shell sort Algorithm.

To understand the working of the shell sort algorithm, let's take an unsorted array. It will be easier to understand the shell sort via an example.

Let the elements of array are -

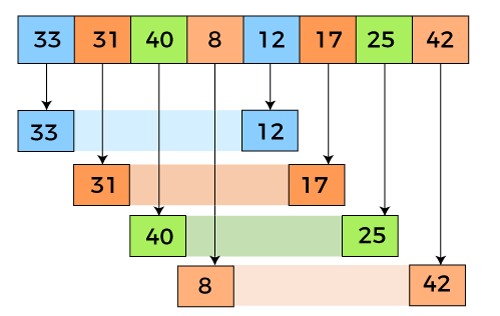


We will use the original sequence of shell sort, i.e., N/2, N/4,....,1 as the intervals.

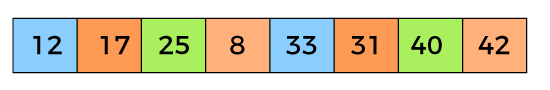
In the first loop, n is equal to 8 (size of the array), so the elements are lying at the interval of 4 (n/2 = 4). Elements will be compared and swapped if they are not in order.

Here, in the first loop, the element at the 0th position will be compared with the element at 4th position. If the 0th element is greater, it will be swapped with the element at 4th position. Otherwise, it remains the same. This process will continue for the remaining elements.

At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.

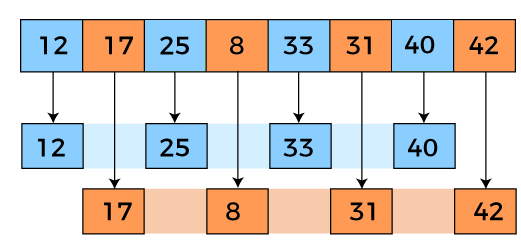


Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -

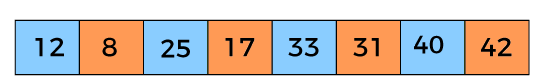


In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.

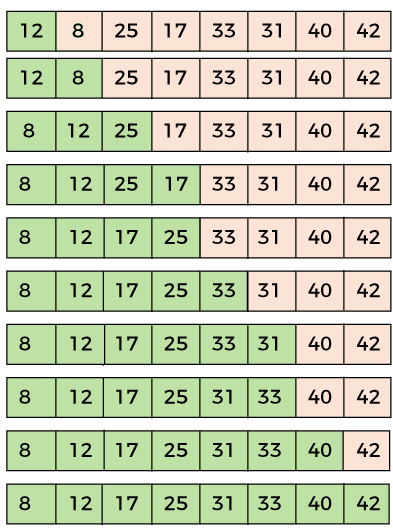
Now, we are taking the interval of **2** to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



Now, we again have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -



In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8. At last, we use the interval of value 1 to sort the rest of the array elements. In this step, shell sort uses insertion sort to sort the array elements.



Now, the array is sorted in ascending order.

Shell sort complexity

Now, let's see the time complexity of Shell sort in the best case, average case, and worst case. We will also see the space complexity of the Shell sort.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n\*logn) |
| **Average Case** | O(n\*log(n)2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of Shell sort is **O(n\*logn).**
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Shell sort is **O(n\*logn).**
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Shell sort is **O(n2).**

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | NO |

* The space complexity of Shell sort is O(1).

Implementation of Shell sort

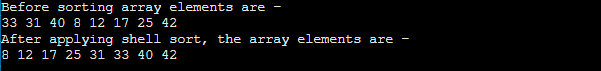
Now, let's see the programs of Shell sort in different programming languages.

**Program:** Write a program to implement Shell sort in C language.

1. #include <stdio.h>
2. /\* function to implement shellSort \*/
3. **int** shell(**int** a[], **int** n)
4. {
5. /\* Rearrange the array elements at n/2, n/4, ..., 1 intervals \*/
6. **for** (**int** interval = n/2; interval > 0; interval /= 2)
7. {
8. **for** (**int** i = interval; i < n; i += 1)
9. {
10. /\* store a[i] to the variable temp and make the ith position empty \*/
11. **int** temp = a[i];
12. **int** j;
13. **for** (j = i; j >= interval && a[j - interval] > temp; j -= interval)
14. a[j] = a[j - interval];
16. // put temp (the original a[i]) in its correct position
17. a[j] = temp;
18. }
19. }
20. **return** 0;
21. }
22. **void** printArr(**int** a[], **int** n) /\* function to print the array elements \*/
23. {
24. **int** i;
25. **for** (i = 0; i < n; i++)
26. printf("%d ", a[i]);
27. }
28. **int** main()
29. {
30. **int** a[] = { 33, 31, 40, 8, 12, 17, 25, 42 };
31. **int** n = **sizeof**(a) / **sizeof**(a[0]);
32. printf("Before sorting array elements are - \n");
33. printArr(a, n);
34. shell(a, n);
35. printf("\nAfter applying shell sort, the array elements are - \n");
36. printArr(a, n);
37. **return** 0;
38. }

**Output**

After the execution of above code, the output will be -

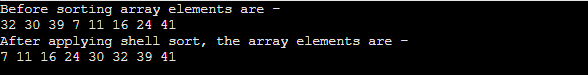


**Program:** Write a program to implement Shell sort in C++.

1. #include <iostream>
2. **using** **namespace** std;
3. /\* function to implement shellSort \*/
4. **int** shell(**int** a[], **int** n)
5. {
6. /\* Rearrange the array elements at n/2, n/4, ..., 1 intervals \*/
7. **for** (**int** interval = n/2; interval > 0; interval /= 2)
8. {
9. **for** (**int** i = interval; i < n; i += 1)
10. {
11. /\* store a[i] to the variable temp and make the ith position empty \*/
12. **int** temp = a[i];
13. **int** j;
14. **for** (j = i; j >= interval && a[j - interval] > temp; j -= interval)
15. a[j] = a[j - interval];
17. // put temp (the original a[i]) in its correct position
18. a[j] = temp;
19. }
20. }
21. **return** 0;
22. }
23. **void** printArr(**int** a[], **int** n) /\* function to print the array elements \*/
24. {
25. **int** i;
26. **for** (i = 0; i < n; i++)
27. cout<<a[i]<<" ";
28. }
29. **int** main()
30. {
31. **int** a[] = { 32, 30, 39, 7, 11, 16, 24, 41 };
32. **int** n = **sizeof**(a) / **sizeof**(a[0]);
33. cout<<"Before sorting array elements are - \n";
34. printArr(a, n);
35. shell(a, n);
36. cout<<"\nAfter applying shell sort, the array elements are - \n";
37. printArr(a, n);
38. **return** 0;
39. }

**Output**

After the execution of the above code, the output will be -

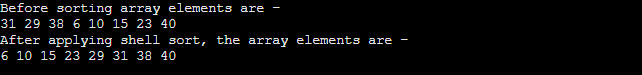


**Program:** Write a program to implement Shell sort in C#.

1. **using** System;
2. **class** ShellSort {
3. /\* function to implement shellSort \*/
4. **static** **void** shell(**int**[] a, **int** n)
5. {
6. /\* Rearrange the array elements at n/2, n/4, ..., 1 intervals \*/
7. **for** (**int** interval = n/2; interval > 0; interval /= 2)
8. {
9. **for** (**int** i = interval; i < n; i += 1)
10. {
11. /\* store a[i] to the variable temp and make the ith position empty \*/
12. **int** temp = a[i];
13. **int** j;
14. **for** (j = i; j >= interval && a[j - interval] > temp; j -= interval)
15. a[j] = a[j - interval];
17. /\* put temp (the original a[i]) in its correct position \*/
18. a[j] = temp;
19. }
20. }
21. }
22. **static** **void** printArr(**int**[] a, **int** n) /\* function to print the array elements \*/
23. {
24. **int** i;
25. **for** (i = 0; i < n; i++)
26. Console.Write(a[i] + " ");
27. }
28. **static** **void** Main()
29. {
30. **int**[] a = { 31, 29, 38, 6, 10, 15, 23, 40 };
31. **int** n = a.Length;
32. Console.Write("Before sorting array elements are - \n");
33. printArr(a, n);
34. shell(a, n);
35. Console.Write("\nAfter applying shell sort, the array elements are - \n");
36. printArr(a, n);
37. }
38. }

**Output**

After the execution of the above code, the output will be -

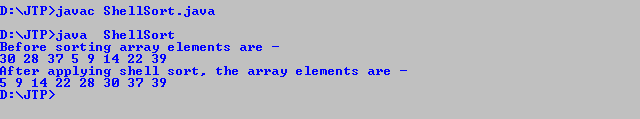


**Program:** Write a program to implement Shell sort in Java.

1. **class** ShellSort {
2. /\* function to implement shellSort \*/
3. **static** **void** shell(**int** a[], **int** n)
4. {
5. /\* Rearrange the array elements at n/2, n/4, ..., 1 intervals \*/
6. **for** (**int** interval = n/2; interval > 0; interval /= 2)
7. {
8. **for** (**int** i = interval; i < n; i += 1)
9. {
10. /\* store a[i] to the variable temp and make
12. the ith position empty \*/
13. **int** temp = a[i];
14. **int** j;
15. **for** (j = i; j >= interval && a[j - interval] >
16. temp; j -= interval)
17. a[j] = a[j - interval];
19. /\* put temp (the original a[i]) in its correct
20. position \*/
21. a[j] = temp;
22. }
23. }
24. }
25. **static** **void** printArr(**int** a[], **int** n) /\* function to print the array elements \*/
26. {
27. **int** i;
28. **for** (i = 0; i < n; i++)
29. System.out.print(a[i] + " ");
30. }
31. **public** **static** **void** main(String args[])
32. {
33. **int** a[] = { 30, 28, 37, 5, 9, 14, 22, 39 };
34. **int** n = a.length;
35. System.out.print("Before sorting array elements are - \n");
36. printArr(a, n);
37. shell(a, n);
38. System.out.print("\nAfter applying shell sort, the array elements are - \n");
39. printArr(a, n);
40. }
41. }

**Output**

After the execution of the above code, the output will be -



So, that's all about the article. Hope the article will be helpful and informative to you.

UNIT-5

Graph

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

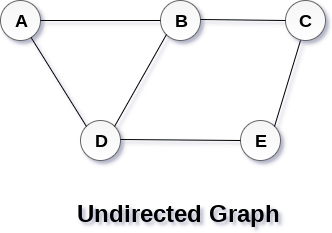
Backward Skip 10sPlay VideoForward Skip 10s

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.

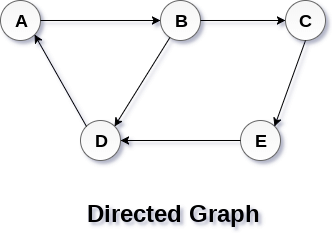


Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.

Simple Path

If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph representation

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

* **Sequential representation** (or, Adjacency matrix representation)
* **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail.

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

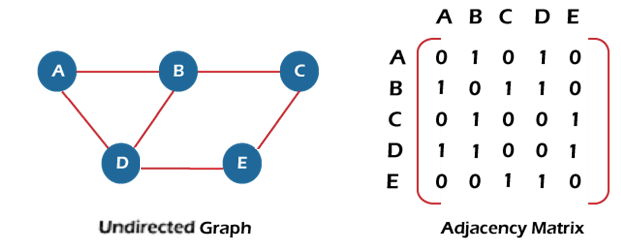
aij = 1 {if there is a path exists from Vi to Vj}

aij = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



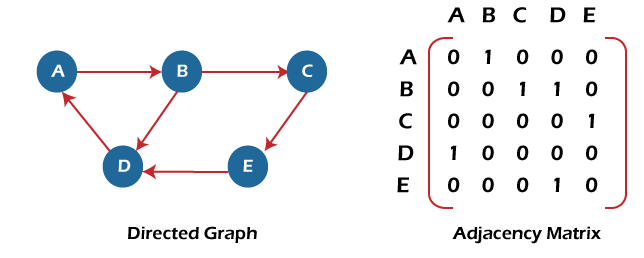
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

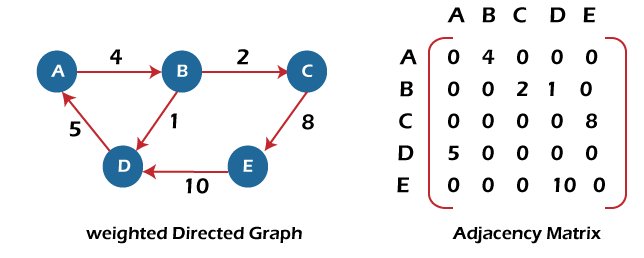
Consider the below-directed graph and try to construct the adjacency matrix of it.



In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

**Adjacency matrix for a weighted directed graph**

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

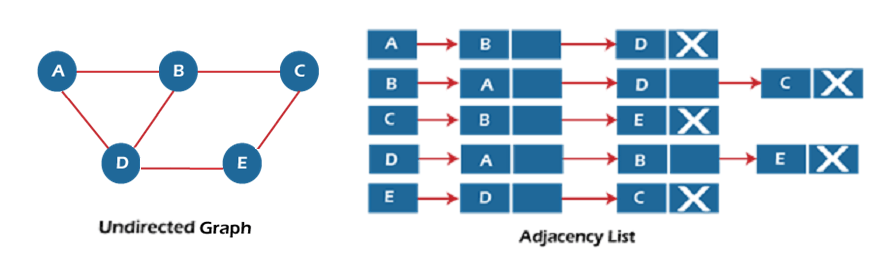
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

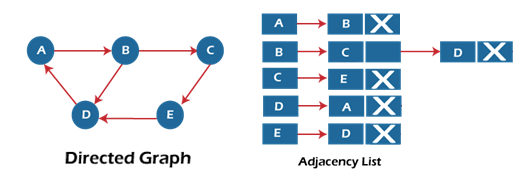


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

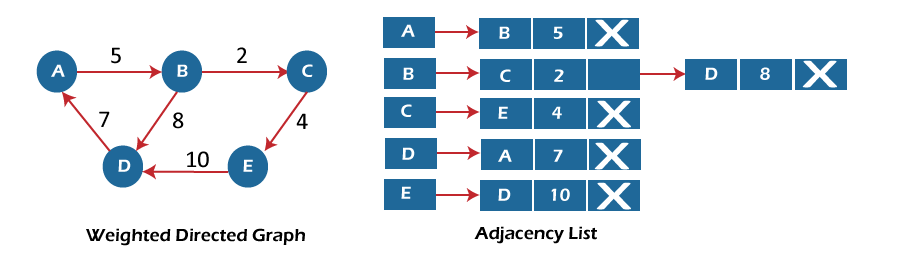
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

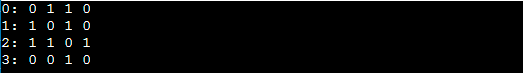
In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

1. /\* Adjacency Matrix representation of an undirected graph in C \*/
3. #include <stdio.h>
4. #define V 4 /\* number of vertices in the graph \*/
6. /\* function to initialize the matrix to zero \*/
7. **void** init(**int** arr[][V]) {
8. **int** i, j;
9. **for** (i = 0; i < V; i++)
10. **for** (j = 0; j < V; j++)
11. arr[i][j] = 0;
12. }
14. /\* function to add edges to the graph \*/
15. **void** insertEdge(**int** arr[][V], **int** i, **int** j) {
16. arr[i][j] = 1;
17. arr[j][i] = 1;
18. }
20. /\* function to print the matrix elements \*/
21. **void** printAdjMatrix(**int** arr[][V]) {
22. **int** i, j;
23. **for** (i = 0; i < V; i++) {
24. printf("%d: ", i);
25. **for** (j = 0; j < V; j++) {
26. printf("%d ", arr[i][j]);
27. }
28. printf("\n");
29. }
30. }
32. **int** main() {
33. **int** adjMatrix[V][V];
35. init(adjMatrix);
36. insertEdge(adjMatrix, 0, 1);
37. insertEdge(adjMatrix, 0, 2);
38. insertEdge(adjMatrix, 1, 2);
39. insertEdge(adjMatrix, 2, 0);
40. insertEdge(adjMatrix, 2, 3);
42. printAdjMatrix(adjMatrix);
44. **return** 0;
45. }

**Output:**

After the execution of the above code, the output will be -



Implementation of adjacency list representation of Graph

Now, let's see the implementation of adjacency list representation of graph in C.

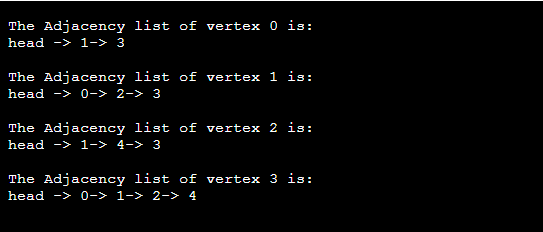
In this program, there is an adjacency list representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

1. /\* Adjacency list representation of a graph in C \*/
2. #include <stdio.h>
3. #include <stdlib.h>
5. /\* structure to represent a node of adjacency list \*/
6. **struct** AdjNode {
7. **int** dest;
8. **struct** AdjNode\* next;
9. };
11. /\* structure to represent an adjacency list \*/
12. **struct** AdjList {
13. **struct** AdjNode\* head;
14. };
16. /\* structure to represent the graph \*/
17. **struct** Graph {
18. **int** V; /\*number of vertices in the graph\*/
19. **struct** AdjList\* array;
20. };

23. **struct** AdjNode\* newAdjNode(**int** dest)
24. {
25. **struct** AdjNode\* newNode = (**struct** AdjNode\*)malloc(**sizeof**(**struct** AdjNode));
26. newNode->dest = dest;
27. newNode->next = NULL;
28. **return** newNode;
29. }
31. **struct** Graph\* createGraph(**int** V)
32. {
33. **struct** Graph\* graph = (**struct** Graph\*)malloc(**sizeof**(**struct** Graph));
34. graph->V = V;
35. graph->array = (**struct** AdjList\*)malloc(V \* **sizeof**(**struct** AdjList));
37. /\* Initialize each adjacency list as empty by making head as NULL \*/
38. **int** i;
39. **for** (i = 0; i < V; ++i)
40. graph->array[i].head = NULL;
41. **return** graph;
42. }
44. /\* function to add an edge to an undirected graph \*/
45. **void** addEdge(**struct** Graph\* graph, **int** src, **int** dest)
46. {
47. /\* Add an edge from src to dest. The node is added at the beginning \*/
48. **struct** AdjNode\* check = NULL;
49. **struct** AdjNode\* newNode = newAdjNode(dest);
51. **if** (graph->array[src].head == NULL) {
52. newNode->next = graph->array[src].head;
53. graph->array[src].head = newNode;
54. }
55. **else** {
57. check = graph->array[src].head;
58. **while** (check->next != NULL) {
59. check = check->next;
60. }
61. // graph->array[src].head = newNode;
62. check->next = newNode;
63. }
65. /\* Since graph is undirected, add an edge from dest to src also \*/
66. newNode = newAdjNode(src);
67. **if** (graph->array[dest].head == NULL) {
68. newNode->next = graph->array[dest].head;
69. graph->array[dest].head = newNode;
70. }
71. **else** {
72. check = graph->array[dest].head;
73. **while** (check->next != NULL) {
74. check = check->next;
75. }
76. check->next = newNode;
77. }
78. }
79. /\* function to print the adjacency list representation of graph\*/
80. **void** print(**struct** Graph\* graph)
81. {
82. **int** v;
83. **for** (v = 0; v < graph->V; ++v) {
84. **struct** AdjNode\* pCrawl = graph->array[v].head;
85. printf("\n The Adjacency list of vertex %d is: \n head ", v);
86. **while** (pCrawl) {
87. printf("-> %d", pCrawl->dest);
88. pCrawl = pCrawl->next;
89. }
90. printf("\n");
91. }
92. }
94. **int** main()
95. {
97. **int** V = 4;
98. **struct** Graph\* g = createGraph(V);
99. addEdge(g, 0, 1);
100. addEdge(g, 0, 3);
101. addEdge(g, 1, 2);
102. addEdge(g, 1, 3);
103. addEdge(g, 2, 4);
104. addEdge(g, 2, 3);
105. addEdge(g, 3, 4);
106. print(g);
107. **return** 0;
108. }

**Output:**

In the output, we will see the adjacency list representation of all the vertices of the graph. After the execution of the above code, the output will be -



Conclusion

Here, we have seen the description of graph representation using the adjacency matrix and adjacency list. We have also seen their implementations in C programming language.

So, that's all about the article. Hope, it will be helpful and informative to you.

BFS algorithm

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

In this article, we will discuss the BFS algorithm in the data structure. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

The applications of breadth-first-algorithm are given as follows -

* BFS can be used to find the neighboring locations from a given source location.
* In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
* BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
* BFS is used to determine the shortest path and minimum spanning tree.
* BFS is also used in Cheney's technique to duplicate the garbage collection.
* It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Enqueue the starting node A and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until QUEUE is empty

**Step 4:** Dequeue a node N. Process it and set its STATUS = 3 (processed state).

**Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

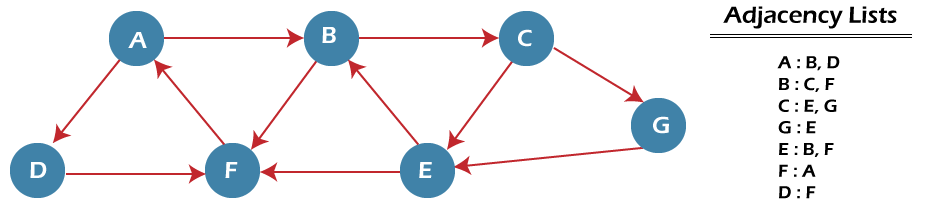
(waiting state)

[END OF LOOP]

**Step 6:** EXIT

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Now, let's start examining the graph starting from Node A.

**Step 1** - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

**Step 2** - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

**Step 3** - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

**Step 4** - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

**Step 5** - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

**Step 5** - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

**Step 6** - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Complexity of BFS algorithm

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is **O(V+E)**, since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

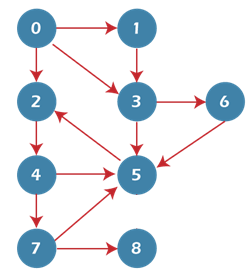
The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

Implementation of BFS algorithm

Now, let's see the implementation of BFS algorithm in java.

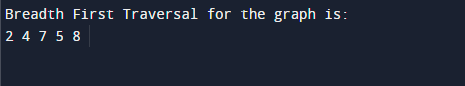
In this code, we are using the adjacency list to represent our graph. Implementing the Breadth-First Search algorithm in Java makes it much easier to deal with the adjacency list since we only have to travel through the list of nodes attached to each node once the node is dequeued from the head (or start) of the queue.

In this example, the graph that we are using to demonstrate the code is given as follows -



1. **import** java.io.\*;
2. **import** java.util.\*;
3. **public** **class** BFSTraversal
4. {
5. **private** **int** vertex;       /\* total number number of vertices in the graph \*/
6. **private** LinkedList<Integer> adj[];      /\* adjacency list \*/
7. **private** Queue<Integer> que;           /\* maintaining a queue \*/
8. BFSTraversal(**int** v)
9. {
10. vertex = v;
11. adj = **new** LinkedList[vertex];
12. **for** (**int** i=0; i<v; i++)
13. {
14. adj[i] = **new** LinkedList<>();
15. }
16. que = **new** LinkedList<Integer>();
17. }
18. **void** insertEdge(**int** v,**int** w)
19. {
20. adj[v].add(w);      /\* adding an edge to the adjacency list (edges are bidirectional in this example) \*/
21. }
22. **void** BFS(**int** n)
23. {
24. **boolean** nodes[] = **new** **boolean**[vertex];       /\* initialize boolean array for holding the data \*/
25. **int** a = 0;
26. nodes[n]=**true**;
27. que.add(n);       /\* root node is added to the top of the queue \*/
28. **while** (que.size() != 0)
29. {
30. n = que.poll();        /\* remove the top element of the queue \*/
31. System.out.print(n+" ");    /\* print the top element of the queue \*/
32. **for** (**int** i = 0; i < adj[n].size(); i++)  /\* iterate through the linked list and push all neighbors into queue \*/
33. {
34. a = adj[n].get(i);
35. **if** (!nodes[a])      /\* only insert nodes into queue if they have not been explored already \*/
36. {
37. nodes[a] = **true**;
38. que.add(a);
39. }
40. }
41. }
42. }
43. **public** **static** **void** main(String args[])
44. {
45. BFSTraversal graph = **new** BFSTraversal(10);
46. graph.insertEdge(0, 1);
47. graph.insertEdge(0, 2);
48. graph.insertEdge(0, 3);
49. graph.insertEdge(1, 3);
50. graph.insertEdge(2, 4);
51. graph.insertEdge(3, 5);
52. graph.insertEdge(3, 6);
53. graph.insertEdge(4, 7);
54. graph.insertEdge(4, 5);
55. graph.insertEdge(5, 2);
56. graph.insertEdge(6, 5);
57. graph.insertEdge(7, 5);
58. graph.insertEdge(7, 8);
59. System.out.println("Breadth First Traversal for the graph is:");
60. graph.BFS(2);
61. }
62. }

**Output**



Conclusion

In this article, we have discussed the Breadth-first search technique along with its example, complexity, and implementation in java programming language. Here, we have also seen the real-life applications of BFS that show it the important data structure algorithm.

So, that's all about the article. Hope, it will be helpful and informative to you.

DFS (Depth First Search) algorithm

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

* DFS algorithm can be used to implement the topological sorting.
* It can be used to find the paths between two vertices.
* It can also be used to detect cycles in the graph.
* DFS algorithm is also used for one solution puzzles.
* DFS is used to determine if a graph is bipartite or not.

Algorithm

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until STACK is empty

**Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)

**Step 5:** Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

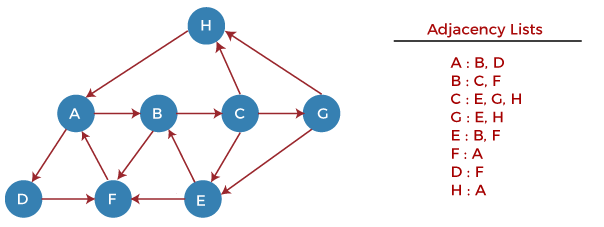
**Step 6:** EXIT

Pseudocode

1. DFS(G,v)   ( v is the vertex where the search starts )
2. Stack S := {};   ( start with an empty stack )
3. **for** each vertex u, set visited[u] := **false**;
4. push S, v;
5. **while** (S is not empty) **do**
6. u := pop S;
7. **if** (not visited[u]) then
8. visited[u] := **true**;
9. **for** each unvisited neighbour w of uu
10. push S, w;
11. end **if**
12. end **while**
13. END DFS()

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Now, let's start examining the graph starting from Node H.

**Step 1** - First, push H onto the stack.

1. STACK: H

**Step 2** - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

**Step 3** - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

**Step 4** - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

**Step 5** - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

**Step 6** - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

**Step 7** - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

**Step 8** - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

**Step 9** - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

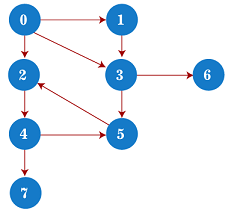
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Implementation of DFS algorithm

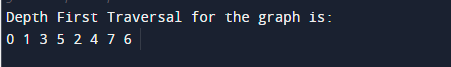
Now, let's see the implementation of DFS algorithm in Java.

In this example, the graph that we are using to demonstrate the code is given as follows -



1. /\*A sample java program to implement the DFS algorithm\*/
3. **import** java.util.\*;
5. **class** DFSTraversal {
6. **private** LinkedList<Integer> adj[]; /\*adjacency list representation\*/
7. **private** **boolean** visited[];
9. /\* Creation of the graph \*/
10. DFSTraversal(**int** V) /\*'V' is the number of vertices in the graph\*/
11. {
12. adj = **new** LinkedList[V];
13. visited = **new** **boolean**[V];
15. **for** (**int** i = 0; i < V; i++)
16. adj[i] = **new** LinkedList<Integer>();
17. }
19. /\* Adding an edge to the graph \*/
20. **void** insertEdge(**int** src, **int** dest) {
21. adj[src].add(dest);
22. }
24. **void** DFS(**int** vertex) {
25. visited[vertex] = **true**; /\*Mark the current node as visited\*/
26. System.out.print(vertex + " ");
28. Iterator<Integer> it = adj[vertex].listIterator();
29. **while** (it.hasNext()) {
30. **int** n = it.next();
31. **if** (!visited[n])
32. DFS(n);
33. }
34. }
36. **public** **static** **void** main(String args[]) {
37. DFSTraversal graph = **new** DFSTraversal(8);
39. graph.insertEdge(0, 1);
40. graph.insertEdge(0, 2);
41. graph.insertEdge(0, 3);
42. graph.insertEdge(1, 3);
43. graph.insertEdge(2, 4);
44. graph.insertEdge(3, 5);
45. graph.insertEdge(3, 6);
46. graph.insertEdge(4, 7);
47. graph.insertEdge(4, 5);
48. graph.insertEdge(5, 2);
50. System.out.println("Depth First Traversal for the graph is:");
51. graph.DFS(0);
52. }
53. }

**Output**



Conclusion

In this article, we have discussed the depth-first search technique, its example, complexity, and implementation in the java programming language. Along with that, we have also seen the applications of the depth-first search algorithm.

So, that's all about the article. Hope it will be helpful and informative to you.

Spanning tree

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

In this article, we will discuss the spanning tree and the minimum spanning tree. But before moving directly towards the spanning tree, let's first see a brief description of the graph and its types.

Graph

A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

* **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
* **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
* **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

Now, let's move towards the topic spanning tree.

What is a spanning tree?

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have **nn-2** number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be **55-2 = 125.**

Applications of the spanning tree

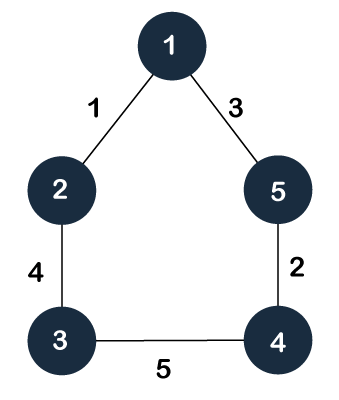
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

* Cluster Analysis
* Civil network planning
* Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

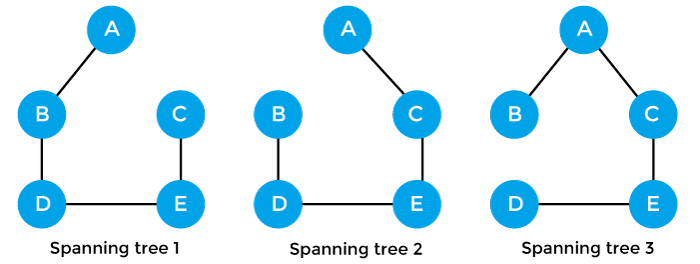
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

* There can be more than one spanning tree of a connected graph G.
* A spanning tree does not have any cycles or loop.
* A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
* A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
* There can be a maximum **nn-2** number of spanning trees that can be created from a complete graph.
* A spanning tree has **n-1** edges, where 'n' is the number of nodes.
* If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

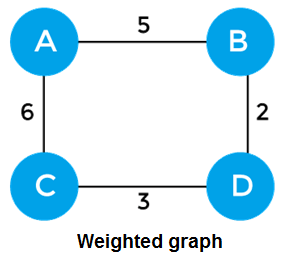
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum Spanning tree

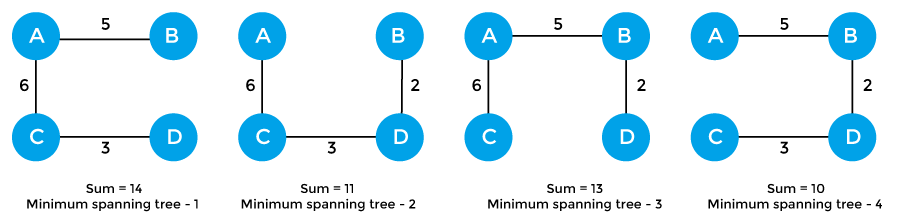
A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

Example of minimum spanning tree

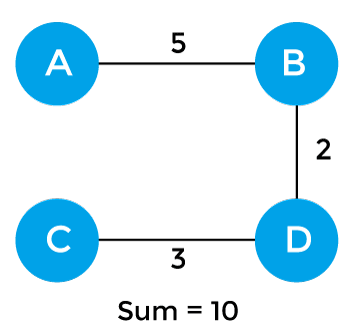
Let's understand the minimum spanning tree with the help of an example.



The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

* Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
* It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

* Prim's Algorithm
* Kruskal's Algorithm

Let's see a brief description of both of the algorithms listed above.

**Prim's algorithm -** It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

To learn more about the prim's algorithm, you can click the below link - <https://www.javatpoint.com/prim-algorithm>

**Kruskal's algorithm -** This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

To learn more about the prim's algorithm, you can click the below link - <https://www.javatpoint.com/kruskal-algorithm>

So, that's all about the article. Hope the article will be helpful and informative to you. Here, we have discussed spanning tree and minimum spanning tree along with their properties, examples, and applications.

Linear Search Algorithm

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

In this article, we will discuss the Linear Search Algorithm. Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Two popular search methods are Linear Search and Binary Search. So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**

The steps used in the implementation of Linear Search are listed as follows -

* First, we have to traverse the array elements using a **for** loop.
* In each iteration of **for loop,** compare the search element with the current array element, and -
  + If the element matches, then return the index of the corresponding array element.
  + If the element does not match, then move to the next element.
* If there is no match or the search element is not present in the given array, return **-1.**

Now, let's see the algorithm of linear search.

Algorithm

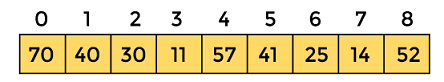
1. Linear\_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
2. Step 1: set pos = -1
3. Step 2: set i = 1
4. Step 3: repeat step 4 while i **<**= n
5. Step 4: if a[i] == val
6. set pos = i
7. print pos
8. go to step 6
9. [end of if]
10. set ii = i + 1
11. [end of loop]
12. Step 5: if pos = -1
13. print "value is not present in the array "
14. [end of if]
15. Step 6: exit

Working of Linear search

Now, let's see the working of the linear search Algorithm.

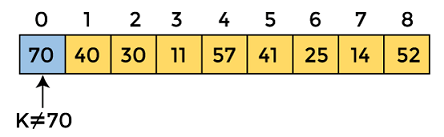
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

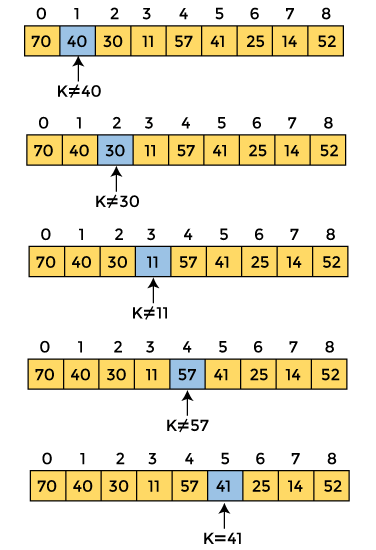


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(1) |
| **Average Case** | O(n) |
| **Worst Case** | O(n) |

* **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
* **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
* **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

The time complexity of linear search is **O(n)** because every element in the array is compared only once.

2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |

* The space complexity of linear search is O(1).

Implementation of Linear Search

Now, let's see the programs of linear search in different programming languages.

**Program:** Write a program to implement linear search in C language.

1. #include <stdio.h>
2. **int** linearSearch(**int** a[], **int** n, **int** val) {
3. // Going through array sequencially
4. **for** (**int** i = 0; i < n; i++)
5. {
6. **if** (a[i] == val)
7. **return** i+1;
8. }
9. **return** -1;
10. }
11. **int** main() {
12. **int** a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array
13. **int** val = 41; // value to be searched
14. **int** n = **sizeof**(a) / **sizeof**(a[0]); // size of array
15. **int** res = linearSearch(a, n, val); // Store result
16. printf("The elements of the array are - ");
17. **for** (**int** i = 0; i < n; i++)
18. printf("%d ", a[i]);
19. printf("\nElement to be searched is - %d", val);
20. **if** (res == -1)
21. printf("\nElement is not present in the array");
22. **else**
23. printf("\nElement is present at %d position of array", res);
24. **return** 0;
25. }

**Output**

Linear Search Algorithm

**Program:** Write a program to implement linear search in C++.

1. #include <iostream>
2. **using** **namespace** std;
3. **int** linearSearch(**int** a[], **int** n, **int** val) {
4. // Going through array linearly
5. **for** (**int** i = 0; i < n; i++)
6. {
7. **if** (a[i] == val)
8. **return** i+1;
9. }
10. **return** -1;
11. }
12. **int** main() {
13. **int** a[] = {69, 39, 29, 10, 56, 40, 24, 13, 51}; // given array
14. **int** val = 56; // value to be searched
15. **int** n = **sizeof**(a) / **sizeof**(a[0]); // size of array
16. **int** res = linearSearch(a, n, val); // Store result
17. cout<<"The elements of the array are - ";
18. **for** (**int** i = 0; i < n; i++)
19. cout<<a[i]<<" ";
20. cout<<"\nElement to be searched is - "<<val;
21. **if** (res == -1)
22. cout<<"\nElement is not present in the array";
23. **else**
24. cout<<"\nElement is present at "<<res<<" position of array";
25. **return** 0;
26. }

**Output**

Linear Search Algorithm

**Program:** Write a program to implement linear search in C#.

1. **using** System;
2. **class** LinearSearch {
3. **static** **int** linearSearch(**int**[] a, **int** n, **int** val) {
4. // Going through array sequencially
5. **for** (**int** i = 0; i < n; i++)
6. {
7. **if** (a[i] == val)
8. **return** i+1;
9. }
10. **return** -1;
11. }
12. **static** **void** Main() {
13. **int**[] a = {56, 30, 20, 41, 67, 31, 22, 14, 52}; // given array
14. **int** val = 14; // value to be searched
15. **int** n = a.Length; // size of array
16. **int** res = linearSearch(a, n, val); // Store result
17. Console.Write("The elements of the array are - ");
18. **for** (**int** i = 0; i < n; i++)
19. Console.Write(" " + a[i]);
20. Console.WriteLine();
21. Console.WriteLine("Element to be searched is - " + val);
22. **if** (res == -1)
23. Console.WriteLine("Element is not present in the array");
24. **else**
25. Console.Write("Element is present at " + res +" position of array");
26. }
27. }

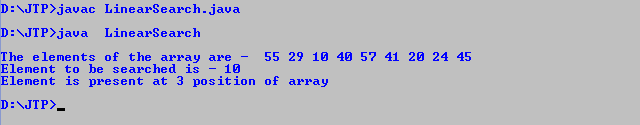
**Output**

Linear Search Algorithm

**Program:** Write a program to implement linear search in Java.

1. **class** LinearSearch {
2. **static** **int** linearSearch(**int** a[], **int** n, **int** val) {
3. // Going through array sequencially
4. **for** (**int** i = 0; i < n; i++)
5. {
6. **if** (a[i] == val)
7. **return** i+1;
8. }
9. **return** -1;
10. }
11. **public** **static** **void** main(String args[]) {
12. **int** a[] = {55, 29, 10, 40, 57, 41, 20, 24, 45}; // given array
13. **int** val = 10; // value to be searched
14. **int** n = a.length; // size of array
15. **int** res = linearSearch(a, n, val); // Store result
16. System.out.println();
17. System.out.print("The elements of the array are - ");
18. **for** (**int** i = 0; i < n; i++)
19. System.out.print(" " + a[i]);
20. System.out.println();
21. System.out.println("Element to be searched is - " + val);
22. **if** (res == -1)
23. System.out.println("Element is not present in the array");
24. **else**
25. System.out.println("Element is present at " + res +" position of array");
26. }
27. }

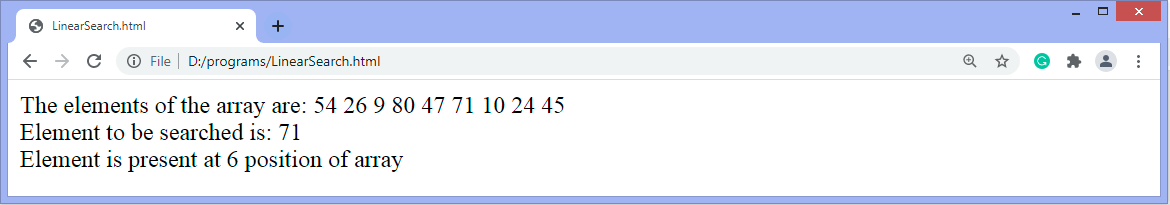
**Output**



**Program:** Write a program to implement linear search in JavaScript.

1. **<html>**
2. **<head>**
3. **</head>**
4. **<body>**
5. **<script>**
6. var a = [54, 26, 9, 80, 47, 71, 10, 24, 45]; // given array
7. var val = 71; // value to be searched
8. var n = a.length; // size of array
9. function linearSearch(a, n, val) {
10. // Going through array sequencially
11. for (var i = 0; i **<** **n**; i++)
12. {
13. if (a[i] == val)
14. return i+1;
15. }
16. return -1
17. }
18. var res = linearSearch(a, n, val); // Store result
19. document.write("The elements of the array are: ");
20. for (i = 0; i **<** **n**; i++)
21. document.write(" " + a[i]);
22. document.write("**<br>**" + "Element to be searched is: " + val);
23. if (res == -1)
24. document.write("**<br>**" + "Element is not present in the array");
25. else
26. document.write("**<br>**" + "Element is present at " + res +" position of array");
27. **</script>**
28. **</body>**
29. **</html>**

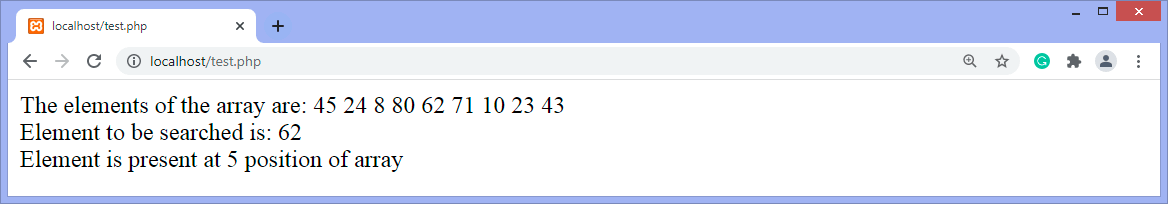
**Output**



**Program:** Write a program to implement linear search in PHP.

1. **<?php**
2. $a = array(45, 24, 8, 80, 62, 71, 10, 23, 43); // given array
3. $val = 62; // value to be searched
4. $n = sizeof($a); //size of array
5. function linearSearch($a, $n, $val) {
6. // Going through array sequencially
7. for ($i = 0; $i **<** $n; $i++)
8. {
9. if ($a[$i] == $val)
10. return $i+1;
11. }
12. return -1;
13. }
14. $res = linearSearch($a, $n, $val); // Store result
15. echo "The elements of the array are: ";
16. for ($i = 0; $i **<** $n; $i++)
17. echo " " , $a[$i];
18. echo "**<br>**" , "Element to be searched is: " , $val;
19. if ($res == -1)
20. echo "**<br>**" , "Element is not present in the array";
21. else
22. echo "**<br>**" , "Element is present at " , $res , " position of array";
23. **?>**

**Output**



So, that's all about the article. Hope the article will be helpful and informative to you.