Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

**How Bubble Sort Works?**

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

**Algorithm**

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**Pseudocode**

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

procedure bubbleSort( list : array of items )

loop = list.count;

for i = 0 to loop-1 do:

swapped = false

for j = 0 to loop-1 do:

/\* compare the adjacent elements \*/

if list[j] > list[j+1] then

/\* swap them \*/

swap( list[j], list[j+1] )

swapped = true

end if

end for

/\*if no number was swapped that means

array is sorted now, break the loop.\*/

if(not swapped) then

break

end if

end for

end procedure return list

programming implementation

#include <stdio.h>

void bubbleSort(int arr[], int n)

{

int i, j, temp;

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

{

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

{

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

{

// swap the elements

temp = arr[j];

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

arr[j+1] = temp;

}

}

}

// print the sorted array

printf("Sorted Array: ");

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

{

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

}

}

int main()

{

int arr[100], i, n, step, temp;

// ask user for number of elements to be sorted

printf("Enter the number of elements to be sorted: ");

scanf("%d", &n);

// input elements if the array

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

{

printf("Enter element no. %d: ", i+1);

scanf("%d", &arr[i]);

}

// call the function bubbleSort

bubbleSort(arr, n);

return 0;

}

**Implementation**

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

**How Selection Sort Works?**

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

**Algorithm**

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

**Pseudocode**

procedure selection sort

list : array of items

n : size of list

for i = 1 to n - 1

/\* set current element as minimum\*/

min = i

/\* check the element to be minimum \*/

for j = i+1 to n

if list[j] < list[min] then

min = j;

end if

end for

/\* swap the minimum element with the current element\*/

if indexMin != i then

swap list[min] and list[i]

end if

end for

end procedure

// C program implementing Selection Sort

# include <stdio.h>

// function to swap elements at the given index values

void swap(int arr[], int firstIndex, int secondIndex)

{

int temp;

temp = arr[firstIndex];

arr[firstIndex] = arr[secondIndex];

arr[secondIndex] = temp;

}

// function to look for smallest element in the given subarray

int indexOfMinimum(int arr[], int startIndex, int n)

{

int minValue = arr[startIndex];

int minIndex = startIndex;

for(int i = minIndex + 1; i < n; i++) {

if(arr[i] < minValue)

{

minIndex = i;

minValue = arr[i];

}

}

return minIndex;

}

void selectionSort(int arr[], int n)

{

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

{

int index = indexOfMinimum(arr, i, n);

swap(arr, i, index);

}

}

void printArray(int arr[], int size)

{

int i;

for(i = 0; i < size; i++)

{

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

}

printf("\n");

}

int main()

{

int arr[] = {46, 52, 21, 22, 11};

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

selectionSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

**How Insertion Sort Works?**

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

**Algorithm**

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

**Step 1** − If it is the first element, it is already sorted. return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the

value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

**Pseudocode**

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure

c++ implementation

#include <stdlib.h>

#include <iostream>

using namespace std;

//member functions declaration

void insertionSort(int arr[], int length);

void printArray(int array[], int size);

// main function

int main()

{

int array[5] = {5, 1, 6, 2, 4, 3};

// calling insertion sort function to sort the array

insertionSort(array, 6);

return 0;

}

void insertionSort(int arr[], int length)

{

int i, j, key;

for (i = 1; i < length; i++)

{

j = i;

while (j > 0 && arr[j - 1] > arr[j])

{

key = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = key;

j--;

}

}

cout << "Sorted Array: ";

// print the sorted array

printArray(arr, length);

}

// function to print the given array

void printArray(int array[], int size)

{

int j;

for (j = 0; j < size; j++)

{

cout <<" "<< array[j];

}

cout << endl;

}

# Merge Sort Algorithm

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.

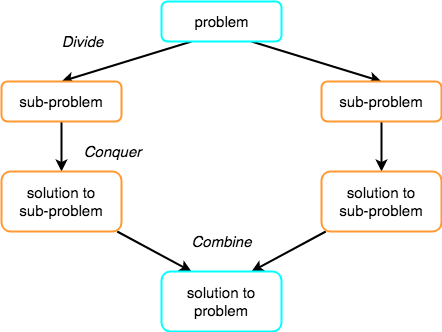
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of O(n2). As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort , on the other hand, runs in O(n\*log n) time in all the cases.

Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of **Divide and Conquer**?



**How Merge Sort Works?**

To understand merge sort, we take an unsorted array as the following −

Unsorted Array

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this −

Merge Sort

Now we should learn some programming aspects of merge sorting.

**Algorithm**

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

**Pseudocode**

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

end while

return c

end procedure

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

// merge sort function

void mergeSort(int a[], int p, int r)

{

int q;

if(p < r)

{

q = (p + r) / 2;

mergeSort(a, p, q);

mergeSort(a, q+1, r);

merge(a, p, q, r);

}

}

// function to merge the subarrays

void merge(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q + 1;

while(i <= q && j <= r)

{

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

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

// function to print the array

void printArray(int a[], int size)

{

int i;

for (i=0; i < size; i++)

{

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

}

printf("\n");

}

int main()

{

int arr[] = {32, 45, 67, 2, 7};

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

printf("Given array: \n");

printArray(arr, len);

// calling merge sort

mergeSort(arr, 0, len - 1);

printf("\nSorted array: \n");

printArray(arr, len);

return 0;

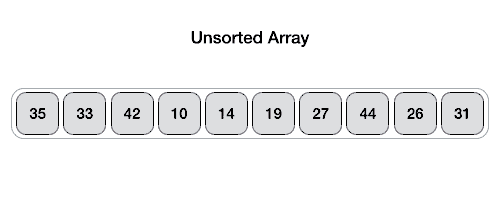
}

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

**Partition in Quick Sort**

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

t is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

**Pivot** element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

**Quick Sort Pivot Algorithm**

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

**Step 1** − Choose the highest index value has pivot

**Step 2** − Take two variables to point left and right of the list excluding pivot

**Step 3** − left points to the low index

**Step 4** − right points to the high

**Step 5** − while value at left is less than pivot move right

**Step 6** − while value at right is greater than pivot move left

**Step 7** − if both step 5 and step 6 does not match swap left and right

**Step 8** − if left ≥ right, the point where they met is new pivot

**Quick Sort Pivot Pseudocode**

The pseudocode for the above algorithm can be derived as −

function partitionFunc(left, right, pivot)

leftPointer = left

rightPointer = right - 1

while True do

while A[++leftPointer] < pivot do

//do-nothing

end while

while rightPointer > 0 && A[--rightPointer] > pivot do

//do-nothing

end while

if leftPointer >= rightPointer

break

else

swap leftPointer,rightPointer

end if

end while

swap leftPointer,right

return leftPointer

end function

**Quick Sort Algorithm**

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

**Step 1** − Make the right-most index value pivot

**Step 2** − partition the array using pivot value

**Step 3** − quicksort left partition recursively

**Step 4** − quicksort right partition recursively

**Quick Sort Pseudocode**

To get more into it, let see the pseudocode for quick sort algorithm −

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

end procedure

// simple C program for Quick Sort

# include <stdio.h>

// to swap two numbers

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

/\*

a[] is the array, p is starting index, that is 0,

and r is the last index of array.

\*/

void quicksort(int a[], int p, int r)

{

if(p < r)

{

int q;

q = partition(a, p, r);

quicksort(a, p, q);

quicksort(a, q+1, r);

}

}

int partition (int a[], int low, int high)

{

int pivot = arr[high]; // selecting last element as pivot

int i = (low - 1); // index of smaller element

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

{

// If current element is smaller than or equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

// function to print the array

void printArray(int a[], int size)

{

int i;

for (i=0; i < size; i++)

{

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

}

printf("\n");

}

int main()

{

int arr[] = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6};

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

// call quickSort function

quickSort(arr, 0, n-1);

printf("Sorted array: n");

printArray(arr, n);

return 0;

}

Counting sort

#include<iostream>

using namespace std;

int k=0; // for storing the maximum element of input array

/\* Method to sort the array \*/

void sort\_func(int A[],int B[],int n)

{

int count[k+1],t;

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

{

//Initialize array count

count[i] = 0;

}

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

{

// count the occurrence of elements u of A

// & increment count[u] where u=A[i]

t = A[i];

count[t]++;

}

for(int i=1;i<=k;i++)

{

// Updating elements of count array

count[i] = count[i]+count[i-1];

}

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

{

// Add elements of array A to array B

t = A[i];

B[count[t]] = t;

// Decrement the count value by 1

count[t]=count[t]-1;

}

}

int main()

{

int n;

cout<<"Enter the size of the array :";

cin>>n;

// A is the input array and will store elements entered by the user

// B is the output array having the sorted elements

int A[n],B[n];

cout<<"Enter the array elements: ";

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

{

cin>>A[i];

if(A[i]>k)

{

// k will have the maximum element of A[]

k = A[i];

}

}

sort\_func(A,B,n);

// Printing the elements of array B

for(int i=1;i<=n;i++)

{

cout<<B[i]<<" ";

}

cout<<"\n";

return 0;

}

## Implementation of Activity Selection Problem Algorithm

Now that we have an overall understanding of the activity selection problem as we have already discussed the algorithm and its working details with the help of an example, following is the C++ implementation for the same.

**Note**: The algorithm can be easily written in any programming language.

#include <bits/stdc++.h>

using namespace std;

#define N 6 // defines the number of activities

// Structure represents an activity having start time and finish time.

struct Activity

{

int start, finish;

};

// This function is used for sorting activities according to finish time

bool Sort\_activity(Activity s1, Activity s2)

{

return (s1.finish< s2.finish);

}

/\* Prints maximum number of activities that can

be done by a single person or single machine at a time.

\*/

void print\_Max\_Activities(Activity arr[], int n)

{

// Sort activities according to finish time

sort(arr, arr+n, Sort\_activity);

cout<< "Following activities are selected \n";

// Select the first activity

int i = 0;

cout<< "(" <<arr[i].start<< ", " <<arr[i].finish << ")\n";

// Consider the remaining activities from 1 to n-1

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

{

// Select this activity if it has start time greater than or equal

// to the finish time of previously selected activity

if (arr[j].start>= arr[i].finish)

{

cout<< "(" <<arr[j].start<< ", "<<arr[j].finish << ") \n";

i = j;

}

}

}

// Driver program

int main()

{

Activity arr[N];

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

{

cout<<"Enter the start and end time of "<<i+1<<" activity \n";

cin>>arr[i].start>>arr[i].finish;

}

print\_Max\_Activities(arr, N);

return 0;

}

## Implementation of Prim's Minimum Spanning Tree Algorithm

Now it's time to write a program in C++ for the finding out minimum spanning tree using prim's algorithm.

#include<iostream>

using namespace std;

// Number of vertices in the graph

const int V=6;

// Function to find the vertex with minimum key value

int min\_Key(int key[], bool visited[])

{

int min = 999, min\_index; // 999 represents an Infinite value

for (int v = 0; v < V; v++) {

if (visited[v] == false && key[v] < min) {

// vertex should not be visited

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// Function to print the final MST stored in parent[]

int print\_MST(int parent[], int cost[V][V])

{

int minCost=0;

cout<<"Edge \tWeight\n";

for (int i = 1; i< V; i++) {

cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";

minCost+=cost[i][parent[i]];

}

cout<<"Total cost is"<<minCost;

}

// Function to find the MST using adjacency cost matrix representation

void find\_MST(int cost[V][V])

{

int parent[V], key[V];

bool visited[V];

// Initialize all the arrays

for (int i = 0; i< V; i++) {

key[i] = 999; // 99 represents an Infinite value

visited[i] = false;

parent[i]=-1;

}

key[0] = 0; // Include first vertex in MST by setting its key vaue to 0.

parent[0] = -1; // First node is always root of MST

// The MST will have maximum V-1 vertices

for (int x = 0; x < V - 1; x++)

{

// Finding the minimum key vertex from the

//set of vertices not yet included in MST

int u = min\_Key(key, visited);

visited[u] = true; // Add the minimum key vertex to the MST

// Update key and parent arrays

for (int v = 0; v < V; v++)

{

// cost[u][v] is non zero only for adjacent vertices of u

// visited[v] is false for vertices not yet included in MST

// key[] gets updated only if cost[u][v] is smaller than key[v]

if (cost[u][v]!=0 && visited[v] == false && cost[u][v] < key[v])

{

parent[v] = u;

key[v] = cost[u][v];

}

}

}

// print the final MST

print\_MST(parent, cost);

}

// main function

int main()

{

int cost[V][V];

cout<<"Enter the vertices for a graph with 6 vetices";

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

{

for(int j=0;j<V;j++)

{

cin>>cost[i][j];

}

}

find\_MST(cost);

return 0;

}