**//Below is a selection sort coding implementation**

#include <iostream>

using namespace std;

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

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void printArray(int array[], int size)

{

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

{

cout << array[i] << " ";

}

cout << endl;

}

void selectionSort(int array[], int size)

{

for (int step = 0; step < size - 1; step++)

{

int min\_idx = step;

for (int i = step + 1; i < size; i++)

{

if (array[i] < array[min\_idx])

min\_idx = i;

}

swap(&array[min\_idx], &array[step]);

}

}

int main()

{

int data[] = {20, 12, 10, 15, 2};

int size = sizeof(data) / sizeof(data[0]);

selectionSort(data, size);

cout << "Sorted array in Acsending Order:\n";

printArray(data, size);

}

**Below is a bubble sort coding implementation**

#include <iostream>

using namespace std;

void bubbleSort(int array[], int size)

{

for (int step = 0; step < size - 1; ++step)

{

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

{

// To sort in descending order, change > to < in this line.

if (array[i] > array[i + 1])

{

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

}

}

}

}

void printArray(int array[], int size)

{

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

{

cout << " " << array[i];

}

cout << "\n";

}

int main()

{

int data[] = {-2, 45, 0, 11, -9};

int size = sizeof(data) / sizeof(data[0]);

bubbleSort(data, size);

cout << "Sorted Array in Ascending Order:\n";

printArray(data, size);

}

**Summary of insertion sort algorithm**

The summary description of how insertion sort algorithm works is this: Start iterating over each item in the data structure (most of the time is an array or vector), starting with the second element in the data structure, check the current element’s value to the element prior to it. For every element’s value in the data set that comes before the current element, if it’s value is less than the current element’s value, move that element up a position in the set and keep cycling through until there is an element whose value is not less than the current element’s value. And place the current element’s value in that spot.

The best times to use this algorithm is:

* When the data set is relatively small
* When items in the data set are semi-sorted already

The time complexities of this algorithm are:

* Worst case performance: O(n²)
* Average case performance: O(n²)
* Best case performance: O(n)