Muzaffar Sharapov

Sorting Algorithms

Sorting array using Selection Sort

While sorting the array, selection sort keeps the two sum arrays intact. A sorted array will come first, followed by an unsorted array. After locating the minvalue pointer to the subsequent element, it finds the lowest value in the array. It sorts in this manner.

The following code snippet checks through the array elements for the smallest value

for (int count = 0; count < size - 1; count++) {   minvalue_pos = count;

for (int var = count + 1; var < size; var++) {   if (array[var] <

array[minvalue_pos]) {   minvalue_pos = var;

}

}

After finding the smallest value within the unsorted array, the position of the value is interchanged to fit the sorted array. *int temp = array[minvalue_pos];*

array[minvalue_pos] = array[count];   array[count] = temp;

Lastly, the sorted array is printed as an output.

    printArr(array, size);

Sorting an array using insertion Sort

In this type of sorting, we attempt to arrange the elements of the array by beginning with the second element of the array, counter = 1, which is the first loop. We compare this value with the first value at counter = 0, if it is smaller, then we take it to the position counter = 0, otherwise, we leave it there. Then, in the second loop, we check the value at counter = 2, and compare it with the value at counter = 1, if it is smaller, we move it to that position, and if it is bigger.

The following lines of codes loop through all the elements of the array starting at the counter = 1 which represents the second value of the array. The procedure goes while moving other elements one step ahead, those which are greater than the target element.

```
for (int counter = 1; counter < size; counter++) {
target = array[counter];        value = counter - 1;
while (value >= 0 && array[value] > target) {
array[value + 1] = array[value];            value =
value - 1;
        }
        array[value + 1] = target;
```

The sorted array is printed by the following line of code

```
printArr(array, size);
```

Sorting array using Shell Sort

The generalized insertion sort algorithm is known as shell sort. The interval between the components to be sorted is gradually decreased after initially sorting elements that are far apart from one another. Depending on the sequence utilized, the distance between the parts is minimized.

The following code snippet shows how the shell sort algorithm utilizes intervals between elements to sort array gradually; *for (int counter = gp; counter < size; counter++) {*

temp = array[counter]; for (value = counter; value >= gp && array[value - gp] > temp; value = value - gp) { array[value] = array[value - gp];

}

array[value] = temp;

}

Sorting array using Bubble Sort

Each pair of adjacent elements in this comparison-based sorting algorithm is compared to each other, and if they are not in the correct order, the elements are swapped. As this algorithm's average and worst-case complexity are both O(n2), where n is the number of items, it is not appropriate for handling huge data sets.

The first two components in a bubble sort are compared to determine which is greater and interchange the position of the values.

for (int counter = 0; counter < size - 1; counter++) { for (int position = 0; position < size - counter - 1; position++) { if (array[position] > array[position + 1]) { int temp =

array[position]; array[position] = array[position + 1]; array[position + 1] = temp;

}

}

Sorting an array using merge Sort

The merging algorithm Sorting with sub-arrays.

Step 1 is to partition the unsorted array into two equal parts.

Step 2 is to sort the two parts, first sorting the first half and then the second.

Step 3 is to merge the two parts to create the sorted array.

The following line of codes illustrates the above procedures respectively:
Creating two temporary sub-arrays for the unsorted array.

int temp_array_1[arr_size_one]; int temp_array_2[arr_size_two];

Copying elements from the original array temporary arrays *for (int counter = 0; counter < arr_size_one; counter++) {*

temp_array_1[counter] = array[l + counter];

}

These lines of codes copy elements to the second array *for*

```
(int counter = 0; counter < arr_size_two; counter++) {

    temp_array_2[counter] = array[mid_index + 1 + counter];

  }
```

These lines of codes merge the two temporary arrays into one block array

```
while (var1 < arr_size_one && var2 < arr_size_two) {      if

(temp_array_1[var1] <= temp_array_2[var2]) {  array[var3] =

temp_array_1[var1]; var1 = var1 + 1; } else { array[var3] =

temp_array_2[var2];

      var2 = var2 + 1;

    }

    var3++;

  }
```

Sorting array using Quick Sort

In this type of sorting, we choose an element from the array to act as the pivot element. We can choose the last element to serve as the pivot, the first element to serve as the pivot, or the middle element to serve as the pivot element. We then divide the selected element into an array around the chosen pivot, position the selected element in the appropriate location, and rearrange the other elements so that smaller elements move below the pivot and bigger elements move above the pivot. In this case, the last element will serve as the pivot key.

```
if (lower_index < higher_index)

  {

int partition_index = ArrPartitioner(array, lower_index, higher_index);

quickSort(array, lower_index, partition_index - 1); quickSort(array,

partition_index + 1, higher_index);

}
```

Below the code I worked on

```
#include <cstdlib>
#include<iostream>

using namespace std;

/*Sorting algorithm methods declaration*/

//Selection sort algo method void
selectionSort(int array[], int size);
```

```
//insertion sort algo method void
insertionSort(int array[], int size);



//shell sort algo method void
shellSort(int array[], int size);

//bubble sort algo method void
bubbleSort(int array[], int size);

//Merge sort algo method void mergeSort(int
array[], int var1, int var2); void
mergeSubArr(int array[], int l, int m, int r);
//Quick Sort algo method int arrPartitioner (int
array[], int first_index, int last_index) ; void
quickSort(int array[], int first_index, int
last_index) ;

//printing array method void
printArr(int array[], int size);

int main(int argc, char** argv) {



   int array[] = {24, 67, 18, 71, 11, 27, 41};

   int size = sizeof (array) / sizeof (array[0]);
   //Sort by selection sort     cout << "Sorted array
By Selection Sort algo: \n";     selectionSort(array,
size);
```

```
//Sort by insertion sort
array[0] = 24;     array[1]
= 67;     array[2] = 18;
array[3] = 71;     array[4]
= 11;     array[5] = 27;
array[6] = 41;
    //reseting the array to the initial unsorted array
cout << "Sorted array By Insertion Sort algo: \n";
insertionSort(array, size);

    //Sort By Shell Sort
    array[0] = 24;     array[1] = 67;     array[2] = 18;
array[3] = 71;     array[4] = 11;     array[5] = 27;
array[6] = 41;//reorganize array to original unsorted array
cout << "Sorted array By Shell Sort algo: \n";
shellSort(array, size);



    //sort By Bubble sort     array[0] = 24;     array[1] = 67;
array[2] = 18;     array[3] = 71;     array[4] = 11;
array[5] = 27;     array[6] = 41;//reorganize array to
original unsorted array     cout << "Sorted array By Bubble
Sort algo: \n";     bubbleSort(array, size);

    //sort by merge sort
array[0] = 24;     array[1]
= 67;     array[2] = 18;
array[3] = 71;     array[4]
```

```
= 11; array[5] = 27;
array[6] =
41;//reorganize array to
original unsorted array
cout << "Sorted array By
Merge Sort Algorithm:
\n";     mergeSort(array, 0,
size - 1);
printArr(array, size);



    /*sort by quick sort*/     array[0] = 24;     array[1] = 67;
array[2] = 18;     array[3] = 71;     array[4] = 11;
array[5] = 27;     array[6] = 41;//reorganize array to
original unsorted array     cout << "Sorted array By Quick
Sort algo: \n";     quickSort(array, 0, size - 1);
printArr(array, size);

   return 0;
}

/*Sort By selection implementation
 Algorithm/ how it works
* This algorithm is simple,  in order to sort an array, we find the smallest value
* and put it at the beginning.(ascending order). we can think that we have to sub array in the
  same array,
* the sorted part, always the first part and the unsorted the second part
* a smallest value is obtained from the unsorted value and put and the end of the sorted part
* 1,3,9,5,4,10
* first round 1,3,9,5,4,10
```

```
* sec round 1,3,4,9,5,10
* third round 1,3,4,5,9,10
* fourth round 1,3,4,5,9,10
 */


void selectionSort(int array[], int size) {
int minvalue_pos;


  for (int count = 0; count < size - 1; count++) {
// check for smallest value in the unsorted array
minvalue_pos = count;

      for (int var = count + 1; var < size; var++) {

         if (array[var] < array[minvalue_pos]) {

            minvalue_pos = var;

         }
      }
      // interchange position       int temp
= array[minvalue_pos];
array[minvalue_pos] = array[count];
array[count] = temp;

      printArr(array, size);

  }
```

```
}

/*INSERTION SORT ALGORITHM TO SORT AN ARRAY*/
//function insertionSort()

void insertionSort(int array[], int size) {     int
target;     int value;     for (int counter = 1; counter
< size; counter++) {
       //we start loop at counter = 1, the second value in the array
target = array[counter];
       //j becomes the position of  previous value, [that is first value if target is the second value]
value = counter - 1;
       //move other elements one step ahead, those which are greater than the target element
while (value >= 0 && array[value] > target) {

          array[value + 1] = array[value];

          value = value - 1;
       }
       array[value + 1] = target;

       printArr(array, size);
    }
}


/*SHELL SORT ALGORITHM TO SORT AN ARRAY*/
//function shellSort()

void shellSort(int array[], int size) {
```

```
    int value;     int temp;     for (int gp = size /
2; gp > 0; gp = gp / 2) {

        for (int counter = gp; counter < size; counter++) {

            temp = array[counter];

            for (value = counter; value >= gp && array[value - gp] > temp; value = value - gp) {

                array[value] = array[value - gp];

            }
            array[value] = temp;

        }

        printArr(array, size);
    }

}

/*BUBBLE SORT ALGORITHM TO SORT AN ARRAY*/
//function bubbleSort()

void bubbleSort(int array[], int size) {

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

        for (int position = 0; position < size - counter - 1; position++) {
```

```
            if (array[position] > array[position + 1]) {

                //interchanging value position
int temp = array[position];
array[position] = array[position + 1];
array[position + 1] = temp;



            }
        }

        printArr(array, size);
    }
}

/*MERGING SUB ARRAYS METHOD*/
//function mergeSubArr()  void mergeSubArr(int array[], int l, int
mid_index, int last_index) {



    int arr_size_one = mid_index - l + 1;
int arr_size_two = last_index - mid_index;

    int var1 = 0; //first subarray initial index     int
var2 = 0; //second subarray initial index     int var3
= l; //mergeSubArr subarray initial index
//temporary arrays     int
```

```
temp_array_1[arr_size_one];     int
temp_array_2[arr_size_two];

   /*copying elements from the original array temporary arrays
    */

    //copying elements to the first array     for (int counter =
0; counter < arr_size_one; counter++) {

        temp_array_1[counter] = array[l + counter];

   }



   //copying elements to the second array     for (int counter =
0; counter < arr_size_two; counter++) {

        temp_array_2[counter] = array[mid_index + 1 + counter];
   }



   //merging the two temporary arrays     while (var1 <
arr_size_one && var2 < arr_size_two) {

        if (temp_array_1[var1] <= temp_array_2[var2]) {

            array[var3] = temp_array_1[var1];
var1 = var1 + 1;

        } else {
```

```
            array[var3] = temp_array_2[var2];
var2 = var2 + 1;
        }
        var3++;
    }

    //Copying remaining elements
while (var1 < arr_size_one) {

        array[var3] = temp_array_1[var1];
var1 = var1 + 1;          var3 = var3 + 1;
    }

    //Copying the temporary array two remaining elements if there are some
while (var2 < arr_size_two) {

        array[var3] = temp_array_2[var2];
var2 = var2 + 1;          var3 = var3 + 1;
    }

    cout << "Merged array:";
printArr(array, last_index);
}
/*MERGE SORT ALGORITHM TO SORT AN ARRAY*/
//function mergeSort() void mergeSort(int array[], int
left_index, int right_index) {     if (left_index < right_index)
{
```

```
        int m = left_index + (right_index - left_index) / 2;

        mergeSort(array, left_index, m);
mergeSort(array, m + 1, right_index);

        mergeSubArr(array, left_index, m, right_index);


    }
}

/*QUICK SORT ALGORITHM TO SORT AN ARRAY*/
//function quickSort()

int ArrPartitioner (int array[], int first_index, int last_index)
{    int pivot = array[last_index];    // choose a pivot element
int var = (first_index - 1);  // Index of a smaller element

    for (int counter = first_index; counter <= last_index- 1; counter++)
    {
        if (array[counter] <= pivot)
        {
var++;

            //swap elements
int tempone = array[var];
array[var] = array[counter];
array[counter] = tempone;
        }
    }
```

```
   //do another swap                  int temptwo
= array[var+1];                 array[var+1] =
array[last_index];
array[last_index] = temptwo;

               //print out array after swapping values
printArr(array, last_index);     return (var + 1);
}




void quickSort(int array[], int lower_index, int higher_index)
{     if (lower_index <
higher_index)
   {
       int partition_index = ArrPartitioner(array, lower_index, higher_index);

       quickSort(array, lower_index, partition_index - 1);



       quickSort(array, partition_index + 1, higher_index);



   }

}
```

```
/* Function for printing an array */ void
printArr(int array[], int size) {
   int i;    for (i = 0; i < size;
i++) {       cout << array[i]
<< " ";
   }

   cout << " " << endl;    cout <<
"**********************" << endl;
}
```