# 

# Department of Computing

**CS 250: Data Structures and Algorithms**

**Class: BSCS-9AB**

**Lab 09: Sorting Algorithms**

**Date: December 07, 2020**

**Time: 10:00 am -1:00pm, 2:00pm – 5:00pm**

# Instructor: Dr. Yasir Faheem

**Lab 9: Sorting Algorithms**

**Introduction**

In this lab, you will implement Selection sort, Bubble Sort, Short Bubble sort and Merge sort algorithms and compare them.

**Objectives**

Objective of this lab is to implement insertion sort and merge sort and compare the running times for both sorting algorithms.

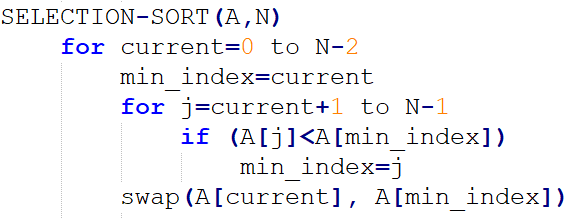
**Tools/Software Requirement**

Visual Studio C++

**Description**

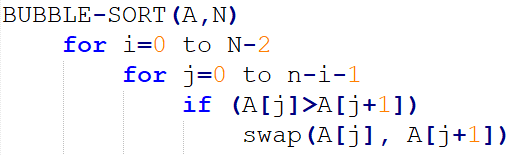
**Selection Sort:**

Selection sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



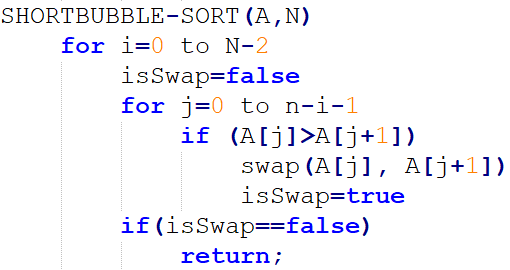
**Bubble Sort:**

Insertion sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



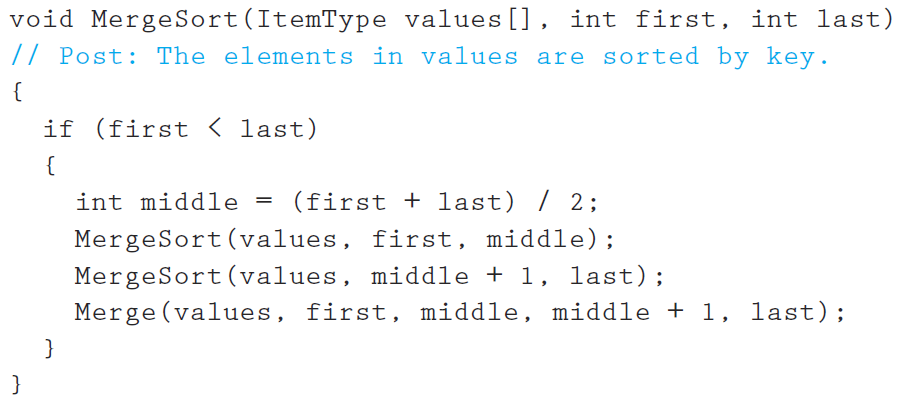
**Short Bubble Sort:**

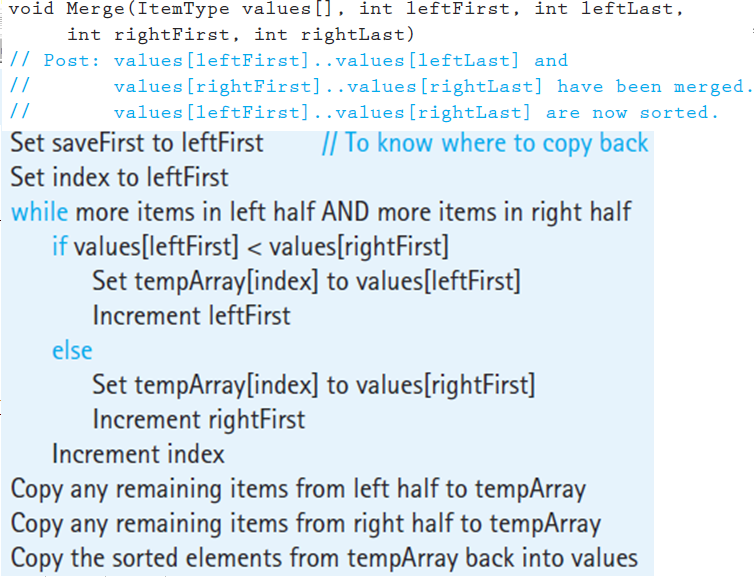
It is a variant of Bubbles sort the best-case complexity of which is Ω (n). Note that the best-case and worst-case complexities of Bubble sort are of order n2.



**Merge Sort:**

Merge sort is another important sorting algorithm that we have seen. Unlike insertion sort, it is not an in-place sorting algorithm. The pseudo code for merge sort is shown below:



****

**Lab Tasks**

**Task 1:**

Implement Selection sort, Bubble sort, Short Bubble sort and Merge sort algorithms in C++.

void selectionSort(int array[3], int n)

{

for (current = i; current <= n - 2; current++)

{//min index is et to 0 at start of loop

min\_index = current;

for (j = (current + 1); j <= n - 1; j++)

{//loop continues from current

//current is boundary between sorted and unsorted part

if (array[min\_index] > array[j])

{//min index stores the minimum value in unsorted part

min\_index = j;

}

}

//swaps at end of one iteration smallest value with start of unsorted part which is current

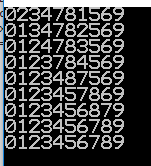
temp = array[current];

array[current] = array[min\_index];

array[min\_index] = temp;

printarray(array, n);

}

}

**Bubble Sort**

class Sorting

{public:

int current=0;

int temp=0;

int j = 0;

void BubbleSort(int array[], int n)

{//Algorithm for sorting

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

{//continues for n no of times

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

{//starts from first element of array and continues till last

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

{//swaps the elements if value at lower index is greter than its preceding one

temp = array[j];

array[j] = array[j+1];

array[j+1] = temp;

}

else

{//else continues

continue;

}

}

printarray(array, n);

}

}

//Function for printing array

void printarray(int array[],int n)

{

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

{

cout << array[i];

}

cout << endl;

}

};

int main()

{

int array[10] = {1,2,3,4,7,8,0,5,6,9};

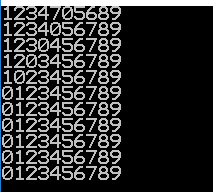
Sorting\* sort = new Sorting();

sort->BubbleSort(array, 10);

sort->printarray(array, 10);

}

**Output**



**Short Bubble Sort:**

void ShortBubbleSort(int array[], int n)

{//Algorithm for sorting

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

{//continues for n no of times

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

{//starts from first element of array and continues till last

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

{//swaps the elements if value at lower index is greter than its preceding one

temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

valswapped = true;

}

else

{//else continues

continue;

}

}

printarray(array, n);

if (!valswapped)

{

break;

}

valswapped = false;

}

}



**Merge Sort**

void MergeSort(int array[],int first,int last)

{

if (first<last)

{

int middle = (last+first)/2;

MergeSort(array,first,middle);

MergeSort(array,middle + 1, last);

Merge(array,first,middle,middle+1,last);

}

else

{

return;

}

}

void Merge(int array[], int leftfirst, int leftlast, int rightfirst, int rightlast)

{

int temparray[max\_size];

int index = leftfirst;

int savefirst = leftfirst;

while ((leftfirst <= leftlast) && (rightfirst <= rightlast))

{

if (array[leftfirst] < array[rightfirst])

{

temparray[index]=array[leftfirst];

leftfirst++;

index++;

}

else

{

temparray[index]=array[rightfirst];

rightfirst++;

index++;

}

}

while (leftfirst <= leftlast)

{ temparray[index] = array[leftfirst];

leftfirst++;

index++;

}

while (rightfirst <= rightlast)

{

temparray[index] = array[rightfirst];

rightfirst++;

index++;

}

for (index = savefirst; index <= rightlast; index++)

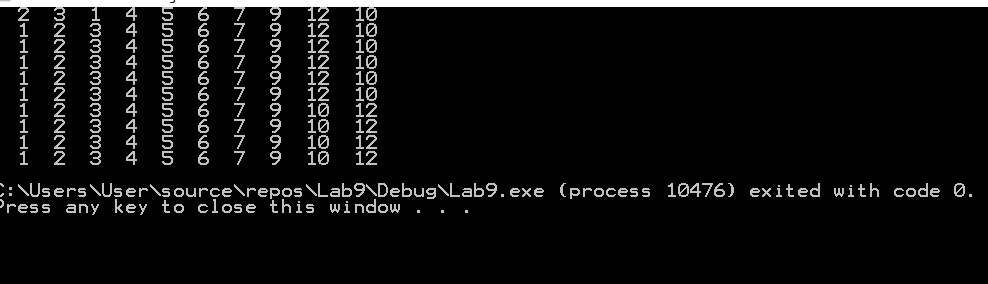
{

array[index]=temparray[index];

}

printarray(array, 10);

}



**Task 2 (average case complexity):**

The next step is to compare the running time of algorithms. Generate arrays of random numbers in the range 1 to 1000 with sizes 100, 1000, and 5000. Compare the running times of the four algorithms on each array. How do they compare? Are the results what you expected, and why? Answer the questions in at the end of the word file.

void AlgorithmComparison()

{

int array100[100];

int arraycopy100[100];

int arraycopy1000[1000];

int arraycopy5000[5000];

int array1000[1000];

int array5000[5000];

//To initialize array of size 100 with random number

for (int i = 0; i < 100; i++)

{

array100[i] = 1 + (rand()%1000);

}

//To initialize array of size 5000 with random number

for (int i = 0; i < 5000; i++)

{

array5000[i] = 1 + (rand()%1000);

}

//To initialize array of size 1000 with random number

for (int i = 0; i < 1000; i++)

{

array1000[i] = 1 + (rand()%1000);

}

//To calculate time for selection sort of an array of size 100

copyarray100(arraycopy100,array100);

auto start = high\_resolution\_clock::now();

selectionSort(arraycopy100, 100);

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by selection sort for size 100 is "<<duration.count() << endl;

//To calculate time for selection sort of an array of size 1000

copyarray1000(arraycopy1000, array1000);

start = high\_resolution\_clock::now();

selectionSort(arraycopy1000, 1000);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by selection sort for size 1000 is " << duration.count() << endl;

//To calculate time for selection sort of an array of size 5000

copyarray1000(arraycopy5000, array5000);

start = high\_resolution\_clock::now();

selectionSort(arraycopy5000, 5000);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by selection sort for size 5000 is " << duration.count() << endl;

//To calculate time for bubble sort of an array of size 100

copyarray100(arraycopy100, array100);

start = high\_resolution\_clock::now();

BubbleSort(arraycopy100, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by bubble sort for size 100 is " << duration.count() << endl;

//To calculate time for bubble sort of an array of size 1000

copyarray1000(arraycopy1000, array1000);

start = high\_resolution\_clock::now();

BubbleSort(arraycopy1000, 1000);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by bubble sort for size 1000 is " << duration.count() << endl;

//To calculate time for bubble sort of an array of size 5000

copyarray5000(arraycopy5000, array5000);

start = high\_resolution\_clock::now();

BubbleSort(arraycopy5000,5000);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by bubble sort for size 5000 is" << duration.count() << endl;

//To calculate time for short bubble sort of an array of size 100

copyarray100(arraycopy100, array100);

start = high\_resolution\_clock::now();

ShortBubbleSort(arraycopy100, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by short bubble sort for size 100 is " << duration.count() << endl;

//To calculate time for short bubble sort of an array of size 1000

copyarray1000(arraycopy1000, array1000);

start = high\_resolution\_clock::now();

ShortBubbleSort(arraycopy1000, 1000);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by short bubble sort for size 1000 is " << duration.count() << endl;

//To calculate time for short bubble sort of an array of size 5000

copyarray5000(arraycopy5000, array5000);

start = high\_resolution\_clock::now();

ShortBubbleSort(arraycopy5000, 5000);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by short bubble sort for size 5000 is " << duration.count() << endl;

//To calculate time for merge bubble sort of an array of size 100

copyarray100(arraycopy100, array100);

start = high\_resolution\_clock::now();

MergeSort(arraycopy100,0,99);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by merge sort for size 100 is " << duration.count() << endl;

//To calculate time for merge bubble sort of an array of size 1000

copyarray1000(arraycopy1000, array1000);

start = high\_resolution\_clock::now();

MergeSort(arraycopy1000, 0, 999);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by merge sort for 1000 is " << duration.count() << endl;

//To calculate time for merge bubble sort of an array of size 5000

copyarray5000(arraycopy5000, array5000);

start = high\_resolution\_clock::now();

MergeSort(arraycopy5000, 0, 4999);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by merge sort for 5000 is " << duration.count() << endl;

}

//method to copy array of size 100

void copyarray100(int arraycopy100[],int array100[])

{

for (int i = 0; i < 100; i++)

{

arraycopy100[i] = array100[i];

}

}

//method to copy array of size 1000

void copyarray1000(int arraycopy1000[1000], int array1000[])

{

for (int i = 0; i < 1000; i++)

{

arraycopy1000[i] = array1000[i];

}

}

//method to copy array of size 5000

void copyarray5000(int arraycopy5000[1000], int array5000[])

{

for (int i = 0; i < 5000; i++)

{

arraycopy5000[i] = array5000[i];

}

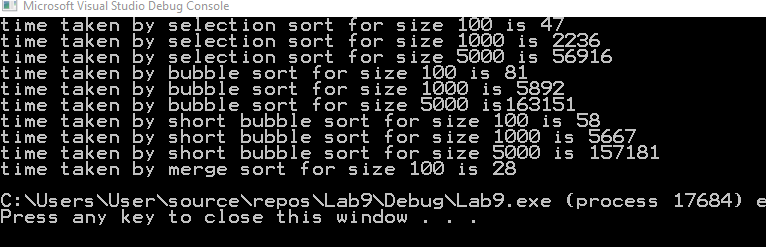
}

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm name** | **Input size 100** | **Input size 1000** | **Input size 5000** |
| **Selection Sort** | 47 | 2236 | 56916 |
| **Bubble Sort** | 81 | 5892 | 163151 |
| **Short Selection** | 58 | 5667 | 157181 |
| **Merge Sort** | 28 | 320 | 5875 |

**Answer**

Time Complexity for selection sort and bubble sort are n2 because of this as input increases ,time also increases to square times. The situation is same for short bubble unless some values are already sorted. Merge sort algorithm has worst case complexity of log2n.Always efficient than above three. For best case scenario, short bubble is more efficient.

**Output**







**Task 3 (best and worst case complexity):**

Now sort the arrays using stl::sort, once in ascending order and then in descending order. Given both sorted arrays as inputs to all the four algorithms and compute their running time. The running time of which algorithm shows most variations based on the structure of the input and why? Answer the questions in at the end of the word file.

**CODE**

void sortarray()

{//built in method used to sort arrays in ascending and descending order

int array[100];

int arraycopy[100];

int arraycopydesc[100];

//Method to generate random numbers for array

RandomArrayGenerator(array);

//built in sort method is used

sort(array, array + 100);

//method call to copy sorted array

copyarray100(arraycopy, array);

//to sort array in descending order

sort(array, array + 100, greater<int>());

copyarray100(arraycopydesc, array);

//To calculate best case complexity for merge sort

auto start = high\_resolution\_clock::now();

MergeSort(arraycopy, 0, 99);

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "best case time complexity by merge sort for size 100 is " << duration.count() << endl;

//To calculate worst complexity for merge sort

start = high\_resolution\_clock::now();

MergeSort(arraycopydesc, 0, 99);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "worst case time complexity by merge sort for size 100 is " << duration.count() << endl;

//copies array sorted in descending to pass as parameter

copyarray100(arraycopydesc, array);

//To calculate worst complexity for selection sort

start = high\_resolution\_clock::now();

selectionSort(arraycopydesc,100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "worst case complexity for selection sort of size 100 is " << duration.count() << endl;

//To calculate best case complexity for selection sort

start = high\_resolution\_clock::now();

selectionSort(arraycopy, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "best case complexity for selection sort of size 100 is " << duration.count() << endl;

//To calculate best case complexity for bubble sort

start = high\_resolution\_clock::now();

BubbleSort(arraycopy, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "best case time complexity for bubble sort for size 100 is " << duration.count() << endl;

//To calculate worst case complexity for bubble sort

copyarray100(arraycopydesc, array);

start = high\_resolution\_clock::now();

BubbleSort(arraycopydesc, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "worst case complexity for Bubble sort of size 100 is " << duration.count() << endl;

//To calculate best case complexity for short bubble sort

start = high\_resolution\_clock::now();

ShortBubbleSort(arraycopy, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "time taken by short bubble sort for size 100 is " << duration.count() << endl;

//To calculate worst case complexity for short bubble sort

copyarray100(arraycopydesc, array);

start = high\_resolution\_clock::now();

ShortBubbleSort(arraycopydesc, 100);

stop = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(stop - start);

cout << "worst case complexity of short bubble sort for size 100 is " << duration.count() << endl;

}

void RandomArrayGenerator(int array[])

{//generates random number array from 1 to 1000

for (int i = 0; i < 100; i++)

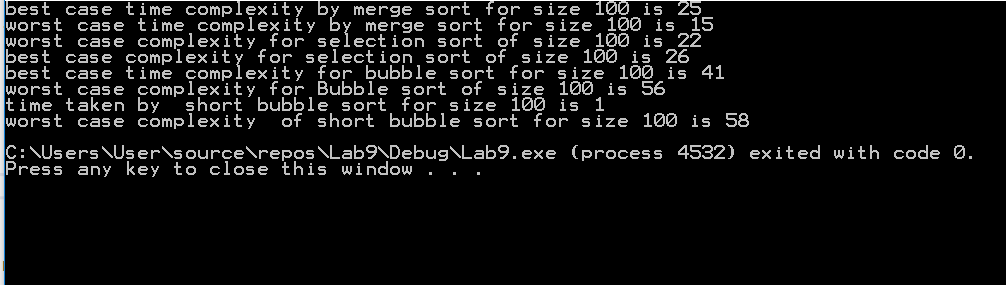
{

array[i] = 1 + (rand() % 1000);

}

}

**Output**



**Answer**

Bubble Sort algorithm shows most variation for sorted and unsorted array as for sorted array the loop will run only one time and then the function returns.

Whereas for all other algorithms the worst case and best case complexity has not much difference for value of 100.

**Deliverables**

Students are required to upload the lab on LMS before deadline.

**Note:** Use proper indentation and comments. Lack of comments and indentation will result in deduction of marks. You will submit your workingcodes in **word document** (do **NOT** take screenshot of code, just copy your code and paste it). The name of word document should follow this format. i.e. **YOUR\_NAME\_Lab#**

# 