Exercise – Insertion Sort

Introduction:

In this exercise we will write the *insertion sort* algorithm and profile its performance.

This tutorial follows on from the *bubble sort* tutorial. If you have not yet completed that tutorial you may want to do that before completing this one. While it is not essential to have completed that tutorial, only the code that needs to be modified will be presented here.

In the next several exercises we will be adding various sorting algorithms to this application and comparing their performance.

Application Setup:

We need to make a few changes from our previous *bubble sort* application so that we can compare the results of both algorithms on the same data.

The main change is the addition of a new array. Before running each sorting algorithm, we will need to copy the original, unsorted values into this new array and then pass this new array to our sorting functions. This is so that we never modify the original unsorted array and can thus use the same data for each function. This is essential to comparing the execution times of each algorithm.

Add a new array variable to the *main()* function as follows:

int main(){

const int size = 100;

// create a large array

int\* values = new int[size];

int\* valuesToSort = new int[size];

srand(time(nullptr));

// and fill it with some ordered data

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

values[i] = rand() % size;

}

high\_resolution\_clock::time\_point t1, t2;

We will pass this new array to our *bubbleSort()* function. But before we can do that, we need to copy the values from our original array into our *valuesToSort* array.

The simple way to do this is using the *memcpy* (memory copy) function:

(following on from the code above)

// Profile Bubble Sort

memcpy(valuesToSort, values, sizeof(int)\*size);

t1 = high\_resolution\_clock::now();

bubbleSort(values, size);

t2 = high\_resolution\_clock::now();

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

std::cout << values[i] << ", ";

}

std::cout << std::endl << std::endl;

std::cout << "bubble sort took " << (t2-t1).count() <<

" nanoseconds" << std::endl;

// pause

std::cout << "press any key to profile Insertion Sort" << std::endl;

std::cin.get();

std::cout << std::endl;

The next step in our program is to run the *insertionSort()* function with the same data.

As before, we copy the original values into our *valuesToSort()* array and pass this array to our *insertionSort()* function.

// Profile Insertion Sort

memcpy(valuesToSort, values, sizeof(int)\*size);

t1 = high\_resolution\_clock::now();

insertionSort(valuesToSort, size);

t2 = high\_resolution\_clock::now();

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

std::cout << valuesToSort[i] << ", ";

}

std::cout << std::endl << std::endl;

std::cout << "Insertion Sort took " << (t2 - t1).count() <<

" nanoseconds" << std::endl;

Finally, don’t forget to delete the newly added *valuesToSort* array before exiting:

delete[] values;

delete[] valuesToSort;

std::cin.ignore(std::cin.rdbuf()->in\_avail());

std::cin.get();

return 0;

}

Exercise:

Complete the insertion sort function. (This function has the same parameters and return value as the *bubble sort* function)

You may want to refer to the pseudo-code and notes included in the lecture slides for this session.

Challenge:

The difference in execution times between the two functions will depend on how sorted the data is.

Modify your program to perform several tests and display the average difference in execution time.