#include <iostream>

#include <iomanip>

#include <stdlib.h>

#include <time.h>

#include <fstream>

#include <ctime>

clock\_t start, finish;

using namespace std;

/\*------------------\*/

int array[50000];

void fillArray(int x)

{

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

{

array[i] = rand() % 1000;

}

}

/\*------------------\*/

void swap(int \*xp, int \*yp)

{

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

} //swap

void insersionSort(int x)

{

int i, key, j;

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

{

key = array[i];

j = i - 1;

//Take an element from the array

//starting at the first element

//that will be the key. Then compare

//the key to each item in the array

//until the item that is next to the

//key is already in sorted order then

//go to the next item as the new key

while (j >= 0 && array[j] > key)

{

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

j = j - 1;

}

array[j + 1] = key;

//Once that loop has finished, do this

//again with the array size minus one

//as many times as the array size minus

//one to end up with a fully sorted array

}

} //insersion

void bubbleSort(int x)

{

//Bubble sort simply works by taking one item,

//comparing it to the next item, and swapping

//them if they are in the wrong order. This is

//the simplest and slowest sorting algorithm

//and needs to traverse the array as many times

//as n^2.

for (int i = 0; i < x - 1; i++) //take an item

{

for (int j = 0; j < x - i - 1; j++) //take the next item

{

if (array[j] > array[j+1]) //compare the two items

{

swap(&array[j], &array[j+1]); //swap if needed

}

}

}

} //bubble

void selectionSort(int x)

{

//Find the minimum element of the array

//and swap it with the first element in

//the array. Then keep on doing this with

//the array being one size smaller and the

//second value now being the first value

//to be swapped with the minimum value.

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

{

int minIndex = i; //min index starts as firt index

for (int j = i + 1; j < x; j++)

{

if (array[j] < array[minIndex])

{

minIndex = j; //locate smallest value in array

}

}

swap(&array[minIndex], &array[i]); //swap

}

} //selection

void heapify(int n, int i)

{

//A Binary Heap is a complete binary tree where

//items are stored in a special order such that

//value in a parenmt node is greater (or smaller)

//than the values in its two children nodes. The

//former is called a max heap and trhe latter is

//called min heap. The heap can be represented

//by binary tree or array.

int largest = i; //root of array is used to initialize largest

int l = 2 \* i + 1; //left child calculated

int r = 2 \* i + 2; //right child calculated

//this assumes that the index starts at 0

//Check if the left child is larger than the root (i)

if (l < n && array[l] > array[largest]) largest = l;

//Check if the right child is larger than the root (i)

if (r < n && array[r] > array[largest]) largest = r;

//Check if the largest data specifier is not equivalent

//to the root (i)

if (largest != i)

{

swap(array[i], array[largest]);

heapify(n, largest);

//Recursively call itself until the full sub-tree is sorted

}

} //heapify

void heapSort(int n)

{

//Call the heapify function to build the heap

for (int i = n / 2 - 1; i >= 0; i--) heapify(n, i);

//Sort each index (element by element)

for (int i = n - 1; i >= 0; i--)

{

swap(array[0], array[i]);

heapify(i, 0);

}

} //heap

int partition (int low, int high)

{

int pivot = array[high]; //pick last element as pivot

int i = (low - 1);

//The goal is to put the pivot in a correct placement

//this is done by moving all elements that are smaller

//than the pivot to the left while putting all elements

//that are greated than the pivot to the right resulting

//in a pivot element that is in a sorted position

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

{

if (array[j] <= pivot)

{

i++;

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

}

}

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

return (i + 1);

} //partition

void quickSort(int low, int high)

{

if (low < high)

{

int pi = partition(low, high); //pi stands for partitioning index

quickSort(low, pi - 1); //Recursively run function with left side of pivot

quickSort(pi + 1, high); //Recursively run function with right side of pivot

}

} //quick

void merge(int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

//Split the main array into left and right arrays

int L[n1], R[n2];

//Fill left and right arrays with data from main array

for (i = 0; i < n1; i++) L[i] = array[l + i];

for (j = 0; j < n2; j++) R[j] = array[m + 1+ j];

i = 0;

j = 0;

k = l;

//Keep splitting the array until left with sets of two

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

array[k] = L[i];

i++;

}

else

{

array[k] = R[j];

j++;

}

k++;

}

//Merge the sorted pairs back into bigger arrays

while (i < n1)

{

array[k] = L[i];

i++;

k++;

} //left

while (j < n2)

{

array[k] = R[j];

j++;

k++;

} //right

} //merge

void mergeSort(int l, int r) //l/r stand for left and right indexes

{

if (l < r)

{

int m = l+(r-l)/2;

mergeSort(l, m);

mergeSort(m+1, r);

merge(l, m, r);

}

} //merge 2

/\*------------------\*/

void doEverything(int aSize)

{

//DATA

double duration;

ofstream of;

of.open("log.txt", fstream::app);

of.width(5);

of.fill('0');

of << endl << aSize;

//INSERSION SORT

fillArray(aSize);

start = clock();

insersionSort(aSize);

finish = clock();

duration = (double)((finish-start)/(double)(CLOCKS\_PER\_SEC));

of << fixed << setfill('0') << setprecision(3) << " " << duration;

//BUBBLE SORT

fillArray(aSize);

start = clock();

bubbleSort(aSize);

finish = clock();

duration = (double)((finish-start)/(double)(CLOCKS\_PER\_SEC));

of << fixed << setfill('0') << setprecision(3) << " " << duration;

//SELECTION SORT

fillArray(aSize);

start = clock();

selectionSort(aSize);

finish = clock();

duration = (double)((finish-start)/(double)(CLOCKS\_PER\_SEC));

of << fixed << setfill('0') << setprecision(3) << " " << duration;

//HEAP SORT

fillArray(aSize);

start = clock();

heapSort(aSize);

finish = clock();

duration = (double)((finish-start)/(double)(CLOCKS\_PER\_SEC));

of << fixed << setfill('0') << setprecision(3) << " " << duration;

//QUICK SORT

fillArray(aSize);

start = clock();

quickSort(0,aSize-1);

finish = clock();

duration = (double)((finish-start)/(double)(CLOCKS\_PER\_SEC));

of << fixed << setfill('0') << setprecision(3) << " " << duration;

//MERGE SORT

fillArray(aSize);

start = clock();

mergeSort(0,aSize-1);

finish = clock();

duration = (double)((finish-start)/(double)(CLOCKS\_PER\_SEC));

of << fixed << setfill('0') << setprecision(3) << " " << duration;

//Close the file

of.close();

}

/\*------------------\*/

int main()

{

srand (time(NULL));

remove("log.txt");

//LABELS

ofstream of;

of.open("log.txt");

of << "DATA SIZE | INSERSION SORT | BUBBLE SORT | SELECTION SORT | HEAP SORT | QUICK SORT | MERGE SORT" << endl;

of.close();

for (int i = 1; i <= 5; i++) doEverything(i\*10000);

}

