1. For this assignment you will write a function that sorts arrays of numbers. Download the following zipped files:

<https://markbowman.org/231/Lab21.zip>

2. Compile and run with the file Data1.txt. Describe the numbers you see. How many are there? Are they in any order?

A picture containing text

Description automatically generated

They aren’t in any particular order. There are 44 numbers within the file.

3. Use the code from Chapter 15 to complete the bubble\_sort() function and call it in your main program, after the call to read\_array(), and before the code to output the array. Example:

Enter file name: ***Data1.txt***

44 values read

-12.4300 -9.9500 -7.1914 -4.2026 -3.2016 0.2790 2.2115 3.1222

3.1824 3.3700 3.4600 4.1417 4.2160 5.2025 5.2724 6.1714

6.2527 6.2715 7.2619 7.2830 9.1327 9.2516 12.2013 13.6300

15.2419 16.2413 23.8500 24.0800 33.7800 42.8900 43.9600 53.4200

57.8000 61.6600 73.6800 73.8900 80.5300 84.4000 93.5300 124.0200

133.1600 213.6300 223.0100 383.6600

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* main()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void main()

{ int i,n;

float x[NMAX];

string fname;

// Get user input

cout << "Enter file name: ";

cin >> fname;

// Fill array

n = read\_array(fname,x,NMAX);

cout << n << " values read" << endl;

//call bubble sort

bubble\_sort(x, n);

// Output list

cout << fixed << setprecision(2);

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

cout << setw(10) << x[i];

cout << endl;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* bubble\_sort()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void bubble\_sort(float map[], int n)

{

int i, j;

float temp;

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

{

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

{

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

{

temp = map[j];

map[j] = map[j + 1];

map[j + 1] = temp;

}

}

}

}

A screenshot of a computer

Description automatically generated with medium confidence

4. Using the global variable count\_comp and count\_swap, add code to count how many times the function compares map elements, and how many times they are swapped. Using the three data files Data1.txt, Data2.txt, and Data3.txt, run the program and fill in the appropriate cells in the table on the next page.

**Data1.txt – Output:**

Graphical user interface

Description automatically generated with low confidence

**Data2.txt – Output:**

Table

Description automatically generated with low confidence

**Data3.txt – Output:**

**Table

Description automatically generated**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* bubble\_sort()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void bubble\_sort(float map[], int n)

{

count\_comp = 0; // initialise to 0

count\_swap = 0; // initialise to 0

int i, j;

float temp;

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

{

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

{

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

{

count\_swap++; // count number of swaps

temp = map[j];

map[j] = map[j + 1];

map[j + 1] = temp;

}

count\_comp++; // count number of comparisons

}

}

}

5. Use the code from Chapter 15 to complete the selection\_sort() function and call it in your main program, replacing the call to bubble\_sort(). Include code to use the global variables to count compares and swaps. Test your function with Data1.txt to make sure it works.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* selection\_sort()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void selection\_sort(float x[],int n)

{

count\_comp = 0; // initialise to 0

count\_swap = 0; // initialise to 0

//selection sory function

int i, j, k;

float temp;

//access each element in list

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

{

//set first element a

// s minimum on first loop

//set minimum as next element on next loop

k = i;

for (j = i; j < n; j++)

{

//compare minimum with each element

//if greater than next element

if (x[k] > x[j])

{

//set minimum to right adjacent element

k = j;

}

//increment comparison count

count\_comp++;

}

//if minimum not in correct place, put minimum in correct place

if (k != i)

{

//swap until in right place

temp = x[i];

x[i] = x[k];

x[k] = temp;

//increment swap count

count\_swap++;

}

}

}

6. Using the three data files Data1.txt, Data2.txt, and Data3.txt, fill in the appropriate cells in the table on the next page.

**Data1.txt – Output:**

Graphical user interface

Description automatically generated with low confidence

**Data2.txt – Output:**

A picture containing text

Description automatically generated

**Data3.txt – Output:**

A picture containing text

Description automatically generated

7. Use the code from Chapter 15 to complete the insertion\_sort() function and call it in your main program, replacing the call to selection\_sort(). Include code to use the global variables to count compares and swaps. Test your function with Data1.txt to make sure it works.

**Data1.txt – Output:**

Graphical user interface

Description automatically generated with low confidence

\*I wanted to confirm what the worst,best, and average case scenarios were for the insertion sort algorithm. I referenced this website courtesy of Louisiana Tech University:

<http://watson.latech.edu/book/algorithms/algorithmsSorting2.html>

According to the above reference:

*Minimum number of insertion sort comparisons  =  N - 1*

*Maximum number of insertion sort comparisons  =  1/2(N2 - N)*

*Average number of insertion sort comparisons  =  1/4(N2 - N)*

Which gives the possible range of insertion sort comparisons as 43 (best case), 473 (average case), and 946 (worst case scenario) for the above case for Data1.txt which has 44 entries.

8. Using the three data files Data1.txt, Data2.txt, and Data3.txt, run the program and fill in the appropriate cells in the table on the next page.

**Data1.txt – Output:**

Graphical user interface

Description automatically generated with low confidence

**Data2.txt – Output:**

**Table

Description automatically generated with low confidence**

**Data3.txt – Output:**

**Text

Description automatically generated with medium confidence**

9. Of the three sorts, which is the most efficient for comparisons, which for swaps?

The bubble sort in this context seems to have on an average basis N\*(N-1) comparisons with N\*(N-1)/2 swaps. In the best case scenario, the bubble sort would have N comparisons and 1 swap.

The selection sort has on average N\*(N-1)/2 to N\*(N+1)/2 comparisons with approximately N swaps that need to be done. In the best cas scenario, only N2 comparisons need to be done and 1 swap has to occur.

The insertion sort (as described above) needs on average N\*(N-1)/4 comparisons to N\*(N-1)/2 in the worst case scenario. The number of swaps that need to be done with the entire insertion sort is in the worst case scenario ~N2 to ~N2/4 it seems in the best case scenario.

The algorithm that is **most efficient for comparisons is the insertion sort** according to these results while the algorithm that is **most efficient for swaps is the selection sort**.

**Final Code:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Lab 21

\* Twymun K. Safford

\* Date Updated: 11/11/2021

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <iomanip>

#include <string>

#include <fstream>

using namespace std;

#define NMAX 1000

int read\_array(string fname,float x[],int n);

void bubble\_sort(float x[],int n);

void selection\_sort(float x[],int n);

void insertion\_sort(float x[],int n);

int count\_comp,count\_swap;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* main()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void main()

{ int i,n;

float x[NMAX];

string fname;

// Get user input

cout << "Enter file name: ";

cin >> fname;

// Fill array

n = read\_array(fname,x,NMAX);

cout << n << " values read" << endl;

//call bubble sort

//bubble\_sort(x, n);

//call selection sort

//selection\_sort(x, n);

//call insertion sort

insertion\_sort(x, n);

cout << "Count compare: " << count\_comp << endl;

cout << "Count swap: " << count\_swap << endl;

// Output list

cout << fixed << setprecision(2);

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

cout << setw(10) << x[i];

cout << endl;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* read\_array()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int read\_array(string fname,float x[],int n)

{ int i;

fstream f;

// Open file

f.open(fname,ios::in);

if(!f.is\_open()) return 0;

// Read from file

i = 0;

while(i<n && !f.eof())

{ f >> x[i];

if(f.good()) i++;

};

// Close and return

f.close();

return i;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* bubble\_sort()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void bubble\_sort(float map[], int n)

{

count\_comp = 0; // initialise to 0

count\_swap = 0; // initialise to 0

int i, j;

float temp;

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

{

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

{

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

{

count\_swap++; // count number of swaps

temp = map[j];

map[j] = map[j + 1];

map[j + 1] = temp;

}

count\_comp++; // count number of comparisons

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* selection\_sort()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void selection\_sort(float x[],int n)

{

count\_comp = 0; // initialise to 0

count\_swap = 0; // initialise to 0

//selection sory function

int i, j, k;

float temp;

//access each element in list

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

{

//set first element a

// s minimum on first loop

//set minimum as next element on next loop

k = i;

for (j = i; j < n; j++)

{

//compare minimum with each element

//if greater than next element

if (x[k] > x[j])

{

//set minimum to right adjacent element

k = j;

}

//increment comparison count

count\_comp++;

}

//if minimum not in correct place, put minimum in correct place

if (k != i)

{

//swap until in right place

temp = x[i];

x[i] = x[k];

x[k] = temp;

//increment swap count

count\_swap++;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* insertion\_sort()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void insertion\_sort(float x[],int n)

{

count\_comp = 0; // initialise to 0

count\_swap = 0; // initialise to 0

//insertion sort function

int i, j, pos;

float temp[NMAX]; //temporary array

//loop through the map

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

{

//find insertion position in x

for (pos = 0; pos < i && temp[pos] < x[i]; pos++);

count\_comp++;

//shift the rest over by one

for (j = i; j > pos; j--)

{

temp[j] = temp[j - 1];

count\_comp++;

count\_swap++;

}

//insert into temp map

temp[pos] = x[i];

}

//copy back into x

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

{

x[i] = temp[i];

}

}

Sort Comparison

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Bubble | | Selection | | Insertion | |
| File | N | Comps | Swaps | Comps | Swaps | Comps | Swaps |
| Data1.txt | 44 | 1892 | 474 | 990 | 42 | 518 | 474 |
| Data2.txt | 252 | 63252 | 16434 | 31878 | 247 | 16691 | 16439 |
| Data3.txt | 1000 | 999000 | 244864 | 500500 | 993 | 246665 | 245665 |
| Order and  Avg Coefficient | | **0.9906 N2** | **0.2495 N2** | **0.5045 N2** | **0.9759 N** | **0.25899 N2** | **0.24978833 N2** |

**Bubble Sort Calculations:**

**Comparisons**

**Data1.txt:**

*Coefficient O(N) = count/N = 1892/44 = 43 N*

*Coefficient O(N2) = count/N2 = 1892/442 =* ***0.977 N2***

**Data2.txt:**

*Coefficient O(N) = count/N = 63252/252 = 251 N*

*Coefficient O(N2) = count/N2 = 63252/2522 =* ***0.996 N2***

**Data3.txt:**

*Coefficient O(N) = count/N = 999000/1000 = 999 N*

*Coefficient O(N2) = count/N2 = 999000/10002 =* ***0.999 N2***

**Average:** (0.999+0.996+0.977)/3 = **0.9906 N2**

**Swaps**

**Data1.txt:**

*Coefficient O(N) = count/N = 474/44 = 10.77 N*

*Coefficient O(N2) = count/N2 = 474/442 =* ***0.2448 N2***

**Data2.txt:**

*Coefficient O(N) = count/N = 16434/252 = 65.21 N*

*Coefficient O(N2) = count/N2 = 16434/2522 =* ***0.2588 N2***

**Data3.txt:**

*Coefficient O(N) = count/N = 244864/1000 = 244.864 N*

*Coefficient O(N2) = count/N2 = 244864/10002 =* ***0.244864 N2***

**Average:** (0.2448+0.2588+0.244864)/3 = **0.2495 N2**

**Selection Sort Calculations:**

**Comparisons**

**Data1.txt:**

*Coefficient O(N) = count/N = 990/44 = 22.5 N*

*Coefficient O(N2) = count/N2 = 990/442 =* ***0.511 N2***

**Data2.txt:**

*Coefficient O(N) = count/N = 31878/252 = 126.5 N*

*Coefficient O(N2) = count/N2 = 31878/2522 =* ***0.502 N2***

**Data3.txt:**

*Coefficient O(N) = count/N = 500500/1000 = 500.5 N*

*Coefficient O(N2) = count/N2 = 500500/10002 =* ***0.5005 N2***

**Average:** (0.511+0.502+0.5005)/3 = **0.5045 N2**

**Swaps**

**Data1.txt:**

*Coefficient O(N) = count/N = 42/44 =* ***0.9545 N***

*Coefficient O(N2) = count/N2 = 42/442 = 0.02 N2*

**Data2.txt:**

*Coefficient O(N) = count/N = 247/252 =* ***0.9801 N***

*Coefficient O(N2) = count/N2 = 247/2522 = 0.00389 N2*

**Data3.txt:**

*Coefficient O(N) = count/N = 993/1000 =* ***0.993 N***

*Coefficient O(N2) = count/N2 = 993/10002 = 0.000993 N2*

**Average:** (0.9545+0.9801+0.993)/3 = **0.9759 N**

**Insertion Sort Calculations:**

**Comparisons**

**Data1.txt:**

*Coefficient O(N) = count/N = 518/44 = 11.772 N*

*Coefficient O(N2) = count/N2 = 518/442 =* ***0.2675 N2***

**Data2.txt:**

*Coefficient O(N) = count/N = 16691/252 = 66.234 N*

*Coefficient O(N2) = count/N2 = 16691/2522 =* ***0.26283 N2***

**Data3.txt:**

*Coefficient O(N) = count/N = 246665/1000 = 246.665 N*

*Coefficient O(N2) = count/N2 = 246665/10002 =* ***0.246665 N2***

**Average:** (0.776995)/3 = **0.25899 N2**

**Swaps**

**Data1.txt:**

*Coefficient O(N) = count/N = 474/44 = 10.772 N*

*Coefficient O(N2) = count/N2 = 474/442 =* ***0.2448 N2***

**Data2.txt:**

*Coefficient O(N) = count/N = 16439/252 = 65.234 N*

*Coefficient O(N2) = count/N2 = 16439/2522 =* ***0.2589 N2***

**Data3.txt:**

*Coefficient O(N) = count/N = 245665/1000 = 245.665 N*

*Coefficient O(N2) = count/N2 = 245665/10002 =* ***0.245665 N2***

**Average:** (0.2448+0.2589+0.245665)/3 = **0.24978833 N2**

The coefficient for a single data file with 20 values and a count of 368 would be:

*Coefficient O(N) = count/N = 368/20 = 18.4 N*

*Coefficient O(N2) = count/N2 = 368/202 = 0.92 N2*

The second value is much closer to 1.00 than the first, so we will use that one.

For each column, calculate the coefficient for each file count, then average them together. Select the running order you think is appropriate and enter it with its coefficient into the bottom row.