Assignment 4: Fun with Languages (30%)

Choose **one** of the following three topics.

Requirements for the assignment are specified at the end of each topic.

1. DOBOSORT - IMPROVED BUBBLESORT

Everyone knows that Bubblesort is incredibly slow. But there were improvements made to this algorithm. In 1980 Wlodzimierz Dobosiewicz implemented an efficient variation of Bubblesort[1]. Dobosiewicz did to Bubblesort what Shell did to insertion sort. The algorithm was "rediscovered" in 1991 by Stephen Lacey and Richard Box, and renamed Combsort[2].

The comparison of values can be the most time consuming part of a sorting algorithm. A Bubblesort requires comparisons proportional to the size of the dataset squared, i.e. N^2. Faster sorting algorithms like This means that a list of 1000 elements could require 1 million comparisons in order to sort it. Quicksort require comparisons on the order of $N(\log_2 N)$. The problem essentially lies with low valued items at the tail end of a list. These items will result in the algorithm slowing, as they move only one element each pass of the algorithm.

Like the Shell sort, the improved Bubblesort increases the gap used in comparisons and exchanges. The improved Bubblesort attempts to eliminate these problems elements that bog down the sort by increasing the gap between items being compared.

TASK

You are provided with a basic algorithm for the improved Bubblesort.

Perform the following:

1. Translate the Modified Bubblesort algorithm to Fortran, and Ada.
2. For each language, modify the code so that a the name of a file containing unsorted integers can be input by the user (a file will be provided for testing). Output the sorted integers to a file named **sortedX.txt** (where X=**F** or **A**, depending on whether the program is Fortran, or Ada respectively).
3. Allow for numbers up to 5 digits in length (i.e. max 99,999), and number sequences up to 50,000 integers in length.

---

[1] Dobosiewicz, W., "An efficient variation of bubble sort", *Information Processing Letters*, 11(1), pp.5-6 (1980)

[2] Lacey, S., Box, R., "A Fast, Easy Sort", *BYTE Magazine*, 16(4), pp.315-318,320 (1991)

4. Test each of the languages using the criteria described below. You will need to test the speed of the algorithms. You will need to also implement Quicksort and traditional Bubblesort in both Fortran and Ada for testing purposes.

In addition, make sure to:
- Improve the usability of the program, i.e. both input and output.
- Document the program appropriately.

TESTING

Test each of the languages for speed. This will require you to set up some form of timing criteria for each language (you should time only the invocation of the **sorting** subroutine). You should briefly discuss your results in the Reflection Report.

The paper by Dobosiewicz includes a table (shown below) which compares run-times of his algorithm against those of the traditional Bubblesort, Shellsort and Quicksort (shown below). For your reflection document, produce a similar table comparing Quicksort, Bubblesort and Modified Bubblesort for n=1000, 10000, and 50000.

Table 1
Running times of various sorting algorithms (times in ms)

n	Quicksort	Shellsort	Bubblesort	Modified Bubblesort
10	0.86	0.99	0.6	0.58
50	5.84	8.62	14.6	4.62
100	13.38	21.28	57.3	11.87
500	87.81	153.70	1456.6	84.80
1000	199.8	367.0		194.9
2000	441.2	858.2		450.5
10000	2696.8			2743.8

NB: The input test files will contain the appropriate number of integers, each on a separate line. The output file should contain the sorted numbers separated by a newline (i.e one number per line).

REFLECTION DOCUMENT

Include a succinct 1 page summary (12-point, single-spaced).

Briefly describe your experiences implementing the algorithm in each of the languages: Fortran, and Ada. Was one better than the other? How did the improved Bubblesort fare? Was it faster than Quicksort in any of the cases? What are the algorithms limitations?

DELIVERABLES

The submission should consist of the following items:

- The reflection report (PDF).
- The code (well documented and styled appropriately of course):
  `dobosort.f95, dobosort.adb`
  `qsort.f95, qsort.adb`
  `bubblesort.f95, bubblesort.adb`
- Both the code and the reflection report should be submitted as a ZIP, TAR, or GZIP file.

REFS

- n/a

2. TEXT STATISTICS IN PASCAL

The Unix system utility `wc` calculates basic statistics of words in files - the number of words, lines and characters. It could be more useful though, more like a text analyzer. When studying languages it is often important to analyze textural material in a statistical manner. Such statistical analysis can be performed in many ways. One example is computing averages, such as the average number of words in a sentence, or the average number of vowels per sentence.

TASK

You are provided with the a program written in Pascal to perform basic text statistics.

Design and implement an Ada program (called **textyzer.adb**) to perform text analysis of a file of human readable English text. The text could contain anywhere from 100 to 1000 words. The analysis would involved calculating the following numerics:

- the number of sentences
- the number of words
- the number of characters (all characters)
- the number of numbers, i.e. items like dates that only contain digits.
- the number of punctuation characters

It would also calculate the following:

- the "average number of words/sentence"
- the "average number of characters per word"
- a histogram representing the word length distribution

The algorithm synopsis is:

- Open a file for reading. Prompt the user for the file name. Perform error checking to make sure the file exists.
- Read that file, processing it to calculate the quantitative values described above.
- Output the statistics to the screen, in an appropriate manner that is easy to read.

Create the following functions (apart from the main function) in your program:

`getFilename()`	This subroutine prompts the user for a filename, checks the existence of the file, and returns that filename to the `main` function. If no file exists with that name, the user is re-prompted.

`analyzeText()`	Take the filename as input and process the text within the file. The function extracts each "word" (a series of characters between two blanks and/or punctuation marks), and then calculates and prints to standard output a series of basic statistics, using functions described below. The output should be similar to that shown in the "Sample Output" below. Invoked by the main program.
`isWord()`	Determine if a piece of text is a word, i.e. contains only alphabetic characters.
`isNumber()`	Determine if a piece of text is a number, i.e. contains only numeric digits.
`printHist()`	Prints a histogram representing the word length distribution. It is easy to do this as a horizontal histogram, harder as a vertical histogram. The example output shows a horizontal histogram.

TESTING

Text 1:

Once every few decades, maybe every century, a nation will produce a hero. An Escoffier, a Muhammad Ali, a Dalai Lama, a Joey Ramone, someone who changes everything about their chosen field, who changes the whole landscape. Life after them is never the same. Martin Picard is such a man. A heretofore unencountered hybrid of rugged outdoorsman, veteran chef, with many years of fine dining experience. Renegade, innovator, he is one of the most influential chefs in North America. He is also a proud Quebecois, and perhaps he more than everyone else has defined for a new generation of Americans and Canadians what that means.
Anthony Bourdain, Parts Unknown 2013.

Here are some stats:

```
Character count (a-z)     : 534
Word count                : 109
Sentence count            : 8
Number count              : 1
Punctuation count         : 21
Characters per word       : 4.9
Words per sentence        : 13.8
```

A histogram of the word length distribution might look something like:

```
Histogram - Word Length Distribution
    1   *********
    2   ************
    3   ***************
    4   ********************
    5   ****************
    6   ********
    7   ************
    8   ****
    9   ******
   10   ****
   11   **
   12
   13   *
   14
   15
   16
   17
   18
   19
   20
```

FURTHER READING

- David V. Moffat, Common Algorithms in Pascal with Programs for Reading (1984)

REFLECTION DOCUMENT

Include a succinct 1 page summary (12-point, single-spaced).

Describe your experiences implementing the algorithm in each of the languages: C, Fortran, Ada. This could include a list of the benefits/limitations of each of the languages.

Also answer the following questions:
- Was there any difference in efficiency? (Show using various test data, in table form). Which language was the slowest and why do you think?
- Were some languages better able to deal with multiplying large numbers?

DELIVERABLES

The submission should consist of the following items:
- The reflection report (PDF).
- The code (well documented and styled appropriately of course): **`textyzer.adb`**
- Both the code and the reflection report should be submitted as a ZIP, TAR, or GZIP file.

```
program trailtext(input, output);
var
   ch: char;
   charsinword : integer;
   wordsinsent : integer;
   totalchars : integer;
   totalwords : integer;
   totalsents : integer;
   longword : integer;
   longsent : integer;
   avgchar : real;
   avgword : real;

begin
   charsinword := 0;
   wordsinsent := 0;
   totalchars := 0;
   totalwords := 0;
   totalsents := 0;
   longword := 0;
   longsent := 0;

   writeln('input text:':29);
   writeln;

   ch := ' ';
   while not eof(input) do
      begin
         while ch in ['.','!','?',' '] do
         begin
            read(ch);
            write(ch)
         end;
   if not( ch in [',',':',';']) then
      charsinword := charsinword + 1;
   while not eoln(input)
         and not (ch in ['.','!','?',' ']) do
      begin
      read(ch);
      write(ch);
      if ch in ['A'..'Z','a'..'z'] then
         charsinword := charsinword + 1
      end;
```

```
44
45    wordsinsent := wordsinsent + 1;
46    totalchars := totalchars + charsinword;
47    if longword < charsinword then
48       longword := charsinword;
49    charsinword := 0;
50
51    if ch in ['.','!','?'] then
52       begin
53       totalsents := totalsents + 1;
54       totalwords := totalwords + wordsinsent;
55       if longsent < wordsinsent then
56          longsent := wordsinsent;
57       wordsinsent := 0
58       end;
59
60    if eoln(input) then
61       begin
62       read(ch);
63       writeln
64    end
65 end;
66
67 avgchar := totalchars /totalwords;
68 avgword := totalwords /totalsents;
69
70 writeln('text statistics:':29);
71 writeln('totals     ':20);
72 writeln('characters':28, totalchars:7);
73 writeln('words     ':28, totalwords:7);
74 writeln('sentences ':28, totalsents:7);
75 writeln('averages':20);
76 writeln('characters/word':28, avgchar:10:2);
77 writeln('words/sentence ':28, avgword:10:2);
78 writeln('maxima':20);
79 writeln('characters/word':28, longword:7);
80 writeln('words/sentence ':28, longsent:7);
81 end.
82
```

3. MEANSORT

Quicksort is very quick, but an alternative based on Quicksort is **Meansort**, which uses the mean value of those elements being partitioned as the pivot. The mean value for use during partitioning step *k* is determined during the scan of a previous step. The first partitioning step simply uses the left element. The algorithm was proposed by Dalia Motzkin in a 1983 issue of Communications of the ACM[3]. This paper describes the algorithm, and compares it with Quicksort.

A year later, the Motzkin added a technical correspondence relating to some typographical errors, and more information on the algorithm[4].

TASK

Given the two papers that relate to **Meansort**, and a program written in Pascal which sorts a mere 11 integers. Perform the following tasks:

1. Implement programs for Meansort in Fortran, Ada, and C.

2. For each language, modify the code so that a the name of a file containing unsorted integers can be input by the user (a series of files will be provided for testing). Output the sorted integers to a file named **sortedX.txt** (where X=**F** or **A**, depending on whether the program is Fortran, or Ada respectively).

3. Implement programs for Quicksort in Fortran, Ada, and C (use any algorithm you like, but keep it consistent between languages).

4. Test each of the languages using the criteria described below.

TESTING

1. Test each of the languages for speed. This will require you to set up some form of timing criteria for each language (you should time only the invocation of the **meansort** subroutine). You should briefly discuss your results in the Reflection Report.

A series of input data sets will be provided:
- A random set of 2000 integers.
- A set of 2000 ordered integers.
- A set of 2000 reverse ordered integers.

---

[3] Motzkin, D., "Meansort", CACM, 26(4), pp.250-251 (1983)

[4] Motzkin, D., "Technical Correspondence", CACM, 27(7), pp. 719-722 (1984)

NB: The input test files will contain integers, each on a separate line. The output file should contain the sorted numbers separated by a newline (i.e one number per line).

REFLECTION REPORT

Include a succinct 1 page summary (12-point, single-spaced).

Briefly describe your experiences implementing the algorithm in each of the languages: Fortran, and Ada. Using the datasets provided, build a table of your results from the testing. Is **Meansort** faster than Quicksort?

DELIVERABLES

The submission should consist of the following items:

- The reflection report (PDF).
- The code (well documented and styled appropriately of course):
  `msort.f95, msort.adb, msort.c`
  `qsort.f95, qsort.adb, qsort.c`
- Both the code and the reflection report should be submitted as a ZIP, TAR, or GZIP file.

```
program msort(input,output);
uses
   Sysutils;
const num = 11;
type numarray = array[1..num] of integer;
var
   i, m, n, mean: integer;
   k: numarray;
   infile: TextFile;

procedure meansort(m,n,mean: integer);
var
     i, j, left_sum, right_sum: integer;
     temp, pmean: integer;

begin
   if m < n then
   begin
      left_sum := 0;
      right_sum := 0;
      i := m;
      j := n;

      while true do
      begin
         while (k[i] < mean) and (i <> j) do
         begin
            left_sum := left_sum + k[i];
            i := i + 1;
         end;
         while (k[j] >= mean) and (i <> j) do
         begin
            right_sum := right_sum + k[j];
            j := j - 1;
         end;
         if (i <> j) then
         begin
            temp := k[i];
            k[i] := k[j];
            k[j] := temp;
         end
         else break;
      end;
```

```
44
45          if i = m then exit;
46          right_sum := right_sum + k[j];
47          pmean := trunc(left_sum / (i-m));
48          meansort(m,i-1,pmean);
49          pmean := trunc(right_sum/(n-j+1));
50          meansort(j,n,pmean);
51       end; {if}
52    end;
53
54    begin
55       Assign(infile,'test1.txt');
56       reset(infile);
57       for i:= 1 to num
58       do begin
59          read(infile, k[i]);
60       end;
61
62       for i := 1 to num do write(k[i]:4);
63       writeln;
64
65       m := 1;
66       n := num;
67
68       if k[m] <= k[n] then
69          mean := k[m]+1
70       else
71          mean := k[m];
72
73       meansort(m,n,mean);
74
75       for i := 1 to num do write(k[i]:4);
76       writeln;
77    end.
78
79
80
81
82
```