CMPT 360: Lab Assignment #2
Quicksort

Brady Coles

last updated: January 3, 2018

Contents

1 Course Goals

This assignment fulfills the following goals:

- a group I language (Python)
- implemented on the Mac OS X platform
- a group II language (Fortran)
- implemented on the Mac OS X platform

2 Problem Description

The following programs implement a version of quicksort for sorting lists of floating point numbers read from files.

3 Sample I/O

The input for these programs are unsorted lists of floating point numbers. These lists are then sorted in increasing order and outputted. For example, if the input was the following numbers:

```
5.35
-1.42
21.7
3.14
-2.718
```

The output would be saved to a file:

```
-2.718
-1.42
3.14
5.35
21.7
```

4 Language Comparison

There are some differences between Python and Fortran that affected the ease with which a sorting algorithm could be implemented. Python was the easier language to implement a sort, and the typeless nature of python variables means that the algorithm can sort any data that can be compared using Python's comparison operators. The only type limiting factor in the Python program is the file input, which casts the input strings into floating point numbers. On the other hand, Fortran is a typed language, so the sorting subroutine only accepts arrays of double precision floats, even though it could sort other data as well.

Another nice development feature is the dynamic lists in Python, which made parsing the input file into something sortable very simple. Fortran, like many languages, has arrays whose lengths need to be specified, so it was necessary to track the size as the input was entered into it.

In terms of implementing the actual sorting algorithm, the languages were very similar. The quicksort algorithm could basically be translated line for line from Python to Fortran.

5 Program Documentation

5.1 Errors and Messages

There are some error cases, they are similar between the Python and Fortran programs.

`File not found: <filename>`
The input file was not found.

`Input was unreadable on a line. Skipped`
The line line_number was skipped in the input file because the input could not be converted.

`Array not sorted.`
The sort failed.

5.2 Problem Solution

The main sorting algorithm can be solved the following way. It is a quicksort with Lomuto pivot, which means the last item is chosen as the pivot.

1. Take the last item as a pivot.
2. Split the rest of the array into values less than the pivot and values greater than the pivot, and put the pivot between them.
3. Repeat the sort on the smaller arrays (less than and greater than arrays) until arrays have 1 or 0 values.

5.3 Pseudocode

This is pseudocode implementation of Lomuto quicksort.

```
def sort(a) \\ where a is an array
    if a has 1 or 0 elements
        return a
    end if
    pivot = a[last]
    split = 0
    for i in a.length - 2
        if a[i] < pivot
            swap a[i] and a[split]
            split = split + 1
        end if
    end for
    return sort(a[start:split-1]) + pivot + sort(a[split:last-1])
end sort
```

5.4 Python Version Documentation

The program can be run from a command line, or imported as a module into other python programs.

5.4.1 Command Line Operation

If the program is run in a command line, it takes two arguments. The format is:

```
<moduleName>.py <inputFile> <outputFile>
```

The arguments are not optional. The first is an input file with items to be sorted. The second is the name to give the output file with the sorted items. Note that the input file must be floating point numbers separated by line breaks.

5.4.2 Module Documentation

Functions		
Name	Arguments	Description
`quicksort`	`a` - A list of any primitive types	Returns a sorted version of the input array, in increasing order.
`checkSorted`	`a` - A list of any primitive types	Returns true if the list is sorted in increasing order, false otherwise.
`readItemsFromFile`	`fileName` - the name of a file	Creates a list of floating point numbers from a file. The file must be formatted with one number per line.
`printArrayToFile`	`a` - An array of any primitive types. `filename` - The name of the file to save.	Saves an array to a text file, with one element per line.
`createRandomFile`	`length` - Number of items `name` - The name of the file to save.	Creates of file with a certain number of randomly generated floats between -10000 and 10000. Used to generate test input.

The module imports `os.path, random, sys`. `os.path` is used to determine if files exist. `random` is used to create test input. `sys` is used to get command line arguments.

5.5 Fortran Version Documentation

Modules are written for Fortran 95, specifically the gfortran compiler. Two modules are provided. The program can also be run from the command line.

5.5.1 Command Line Operation

If the program is run in a command line, it takes two arguments. The format is:

```
<programName> <inputFile> <outputFile>
```

The arguments are not optional. The first is an input file with items to be sorted. The second is the name to give the output file with the sorted items. Note that the input file must be floating point numbers separated by line breaks.

5.5.2 Module Documentation

There are two modules, one is dedicated to sorting, and the other to IO.

5.5.3 quicksort Module

The module `quicksort` is used for sorting.

Functions		
Name	Arguments	Description
`check`	`a` - An array of double precision floats. `s, e` - The start and end indexes to check if sorted.	Returns true if the array is sorted in increasing order, false otherwise.
Subroutines		
`quicksort`	`a` - An array of double precision floats. `s, e` - The start and end indexes to include in the sort.	Sorts the array `a` in place, in increasing order.

5.5.4 sortIO Module

The module `sortIO` is used for file parsing and output.

Functions		
Name	Arguments	Description
`readFile`	`fn` - The name of a file, a string.	Creates and returns an array of double precision floating point numbers from a file. The file must be formatted with one number per line.
Subroutines		
`writeFile`	`a` - An array of double precision floats. `fn` - Name of output file.	Saves an array to a text file, with one element per line.

6 Program Listing

6.1 Python Listing

```
#!/usr/env/bin python3

# Author: Brady Coles
# Quicksort Program
import os.path
import random
import sys

# Reads a file, and takes a float from each line and adds it to a list.
# Returns the list.
def readItemsFromFile(fileName):
    if not os.path.isfile(fileName):
        print("File not found: " + fileName)
        return [];
    f = open(fileName, 'r')
    array = []
    for line in f.readlines():
        try:
            array.append(float(line))
        except ValueError:
            print("Input was unreadable on a line. Skipped")
    f.close()
    print("Array read from file")
    return array

# Quicksort using Lomuto partitioning (take last item as pivot).
def quicksort(a):
    # Base case, 1 or 0 elements to sort
    if len(a) <= 1:
        return a;
    pivot = a[len(a) - 1]
    split = 0
    for i in range(len(a) - 1):
        if a[i] <= pivot:
            a[i], a[split] = a[split], a[i]
            split += 1
    a[split], a[len(a) - 1] = pivot, a[split]
```

```
38      # Recursive call
39      a[:split] = quicksort(a[:split])
40      a[split+1:] = quicksort(a[split+1:])
41      return a
42
43  # Checks to see if the elements in an array are sorted in
44  # increasing order.
45  def checkSorted(a):
46      for i in range(1, len(a)):
47          if a[i] < a[i-1]:
48              return False
49      return True
50
51  # Creates a file with name filename and puts each element
52  # of array on a line.
53  def printArrayToFile(a, filename):
54      f = open(filename, 'w')
55      for i in a:
56          f.write(str(i) + '\n')
57      print("Write finished.")
58
59  # Create a file with random, uniformly distributed floats between
60  # -10000 and 10000.
61  # length is number of items, name is file name.
62  def createRandomFile(length, name):
63      f = open(name, 'w')
64      for i in range(length):
65          f.write('' + str(random.uniform(-10000, 10000)) + '\n')
66      f.close()
67      print("Done create file")
68
69  # Run a sort on an input file and save to an output file.
70  def main():
71      if (len(sys.argv) == 3):
72          inFile = sys.argv[1]
73          outFile = sys.argv[2]
74          array = quicksort(readItemsFromFile(inFile))
75          if checkSorted(array):
76              print("Array is sorted.")
77          else:
78              print("Array not sorted.")
79          printArrayToFile(array, outFile)
80      else:
81          print("Invalid arguments. Enter an input and output file:\nHW2.py <inFile> <outFile>")
82
83  # Only run main if this is the main program
84  # If this module is imported, this does not run.
85  if __name__ == "__main__":
86      main()
```

6.2 Fortran Listing

```
1  ! Author: Brady Coles
2  ! Quicksort Program
```

```
3
4   ! This module has a sort subroutine, and a function to check if an array is
5   ! sorted. Only works on Double Precision floats.
6   MODULE quicksort
7       IMPLICIT NONE
8   CONTAINS
9       ! Sorts array a with a quicksort, between (and including) elements s and e
10      RECURSIVE SUBROUTINE sort(a, s, e)
11          DOUBLE PRECISION, DIMENSION (:), INTENT(inout) :: a
12          INTEGER, INTENT(in) :: s, e
13          INTEGER :: split, i
14          DOUBLE PRECISION :: temp
15
16          ! Base case, 1 or 0 elements to sort
17          IF (e - s < 1) THEN
18              RETURN
19          END IF
20
21          split = s
22          DO i = s, e - 1
23              IF (a(i) <= a(e)) THEN
24                   temp = a(i)
25                   a(i) = a(split)
26                   a(split) = temp
27                   split = split + 1
28              END IF
29          END DO
30          temp = a(split)
31          a(split) = a(e)
32          a(e) = temp
33          ! Recursive call
34          call sort(a, s, split-1)
35          call sort(a, split+1, e)
36      END SUBROUTINE sort
37      ! Checks if elements in array a between (and including) elements s and e
38      ! are sorted in increasing order.
39      FUNCTION check(a, s, e)
40          DOUBLE PRECISION, DIMENSION (:), INTENT(in) :: a
41          INTEGER, INTENT(in) :: s, e
42          LOGICAL :: check
43          INTEGER :: i
44
45          check = .TRUE.
46          DO i = s + 1, e
47              IF (a(i) < a(i-1)) THEN
48                  PRINT *, 'Array not sorted.'
49                  check = .FALSE.
50                  RETURN
51              END IF
52          END DO
53          PRINT *, 'Array is sorted.'
54      END FUNCTION check
55  END MODULE quicksort
56
```

```
57  ! This module can read a file to create an array and make a file from an
58  ! array. Only works on Double Precision floats.
59  MODULE sortIO
60      IMPLICIT NONE
61  CONTAINS
62      ! Reades a file named fn and puts each line in an array.
63      FUNCTION readfile(fn)
64          CHARACTER(len=*), INTENT(in) :: fn
65          DOUBLE PRECISION :: x
66          INTEGER :: arrsize, i, ios
67          DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: readfile
68          DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: temp
69          LOGICAL :: file_exists
70
71          ! Check that file exists
72          INQUIRE(file=fn, exist=file_exists)
73          IF (.NOT. file_exists) THEN
74              PRINT *, 'File not found: ', fn
75              STOP 1
76          END IF
77
78          i = 1
79          arrsize = 1; allocate(readfile(1))
80
81          OPEN (1, file=fn)
82          DO
83              READ(1,*,IOSTAT=ios) x
84              IF (ios > 0) THEN
85                  PRINT *, 'Input was unreadable on a line. Skipped.'
86              ELSE IF (ios < 0) THEN
87                  ! End of file
88                  EXIT
89              ELSE
90                  IF (i > SIZE(readfile)) THEN
91                      ! Dynamically resize readfile when it gets full
92                      arrsize = SIZE(readfile)
93                      arrsize = arrsize + arrsize
94                      ALLOCATE(temp(arrsize))
95                      temp(:SIZE(readfile)) = readfile
96                      CALL MOVE_ALLOC(temp, readfile)
97                  END IF
98                  readfile(i) = x
99                  i = i + 1
100             END IF
101         END DO
102         CLOSE(1)
103         readfile = readfile(:i-1) ! Resize to only include filled elements
104         PRINT *, 'Array read from file'
105     END FUNCTION readfile
106     ! Creates a file with filename fn and puts array a into it, one element
107     ! per line.
108     SUBROUTINE writefile(a, fn)
109         DOUBLE PRECISION, DIMENSION (:), INTENT(in) :: a
110         CHARACTER(len=*), INTENT(in) :: fn
```

```
111         INTEGER :: i
112         OPEN (2, file=fn, status='REPLACE')
113         DO i=1, SIZE(a)
114             WRITE(2,*) a(i)
115         END DO
116         CLOSE(2)
117         PRINT *, 'Write finished.'
118     END SUBROUTINE writefile
119 END MODULE sortIO
120
121 ! Command line program
122 PROGRAM test
123 USE quicksort
124 USE sortIO
125 DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: arr
126 CHARACTER(len=100) :: infn, outfn
127 IF (COMMAND_ARGUMENT_COUNT() == 2) THEN
128     call GET_COMMAND_ARGUMENT(1, infn)
129     call GET_COMMAND_ARGUMENT(2, outfn)
130     arr = readfile(infn)
131     call sort(arr, 1, SIZE(arr))
132     IF (.NOT. check(arr, 1, SIZE(arr))) print *, 'Program failed.'
133     call writefile(arr, outfn)
134 ELSE
135     PRINT *, 'Invalid arguments. Enter an input and output file:'
136     PRINT *, 'lab2.out <inFile> <outFile>'
137 END IF
138 END PROGRAM
139
140
```

7 Input Output Data Files

Sample input and output can be found in the SampleIO folder. There are examples ranging from 20 items to a hundred thousand, plus a million floats, and also 10 million integers with only python output. Some is shown here. The input files were mostly created with the createRandomFile function in the python program.

7.1 20 Elements

Input file

```
-85.70901389618047
52.605743562402324
-86.07603902666081
13.998222568620548
59.99322360450009
43.12829306686615
84.88276456841456
-4.4795882911088825
-45.45427232305759
78.367595867526
-8.086168547353452
-94.23602094029373
-32.887059794593014
-95.72467661988084
-14.69609863152688
-84.00103531452005
50.237480017220776
-52.03746382671133
-81.14875100365455
-39.26645700307769
```

Python output

```
-95.72467661988084
-94.23602094029373
-86.07603902666081
-85.70901389618047
-84.00103531452005
-81.14875100365455
-52.03746382671133
-45.45427232305759
-39.26645700307769
-32.887059794593014
-14.69609863152688
-8.086168547353452
-4.4795882911088825
13.998222568620548
43.12829306686615
50.237480017220776
52.605743562402324
59.99322360450009
78.367595867526
84.88276456841456
```

Fortran Output

```
-95.7246780
-94.2360229
-86.0760422
-85.7090149
-84.0010376
-81.1487503
-52.0374641
-45.4542732
-39.2664566
-32.8870583
-14.6960983
-8.08616829
-4.47958851
 13.9982224
 43.1282921
 50.2374802
 52.6057434
 59.9932251
 78.3675995
 84.8827667
```

7.2 Bad Input

For a test input with invalid lines, this is the result.

Input:

```
8.658
9.5476457
Hello
7
**
4.5
```

Python Console:

```
Input was unreadable on a line. Skipped
Input was unreadable on a line. Skipped
Array read from file
Array is sorted.
Write finished.
```

Python Output:

```
4.5
7.0
8.658
9.5476457
```

Fortran Console:

```
Input was unreadable on a line. Skipped.
Input was unreadable on a line. Skipped.
Array read from file
Array is sorted.
Write finished.
```

Fortran Output:

```
4.5000000000000000
7.0000000000000000
8.6579999999999995
9.5476457000000003
```