Chapter 38
Sorting and Searching

The Analytical Engine weaves algebraic patterns, just as the Jacquard loom weaves flowers and leaves.

Ada Lovelace

Aims
We look at a number of sorting and searching examples:

- three numeric sorting examples, using a recursive algorithm, a non recursive algorithm and a parallelised subroutine from the Nag library
  - timing details for our generic serial Quicksort algorithm for five of the numeric kind types
  - timing details of the Netlib serial non recursive Quicksort for 32 bit integers, 32 bit reals and 64 bit reals
  - a comparison of the timing of the above two sorting algorithms
  - the Nag SMP sorting routine m01caf for 64 bit reals
  - timing details of the parallel Nag sorting subroutine
- Sorting an array of a derived type
- A searching example

38.1 Example 1: Generic Recursive Quicksort Example with Timing Details

This example has several components

- a module called `precision_module` from Chap. 21
- a module called `integer_kind_module` from Chap. 25

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_38

- a timing module
- the generic Quicksort module from Chap. 25
- a main program to provide the timing information

Here is the source code for the main program. The source code for the other modules is the same as in earlier chapters.

```
include 'integer_kind_module.f90'
include 'precision_module.f90'
include 'sort_data_module.f90'
include 'timing_module.f90'

program ch3801

  use sort_data_module
  use timing_module

  implicit none
  integer, parameter :: n = 100000000
  character *12 :: nn = '100,000,000'
  character *80 :: report_file_name = &
    'ch3601.report'

  real (sp), allocatable, dimension (:) :: x_sp
  real (dp), allocatable, dimension (:) :: x_dp
  real (qp), allocatable, dimension (:) :: x_qp
  integer (i32), allocatable, dimension (:) :: &
    y_i32
  integer (i64), allocatable, dimension (:) :: &
    y_i64

  integer :: allocate_status = 0

  character *20, dimension (5) :: heading1 = [ &
    '  32 bit real', '  32 bit int ', &
    '  64 bit real', '  64 bit int ', &
    ' 128 bit real' ]

  character *20, dimension (3) :: heading2 = [ &
    '      Allocate ', '      Random   ', &
    '      Sort     ' ]

  print *, 'Program starts'
  print *, 'N = ', nn
  call start_timing()
```

```
  open (unit=100, file=report_file_name)

  print *, heading1(1)

  allocate (x_sp(1:n), stat=allocate_status)
  if (allocate_status/=0) then
    print *, &
      ' Allocate failed. Program terminates'
    stop 10
  end if

  print 100, heading2(1), time_difference()
100 format (a20, 2x, f8.3)
  call random_number(x_sp)
  print 100, heading2(2), time_difference()
  call sort_data(x_sp, n)
  print 100, heading2(3), time_difference()
  write (unit=100, fmt='(a)') &
    ' First 10 32 bit reals'
  write (unit=100, fmt=110) x_sp(1:10)
110 format (5(2x,e14.6))

  print *, heading1(2)

  allocate (y_i32(1:n), stat=allocate_status)
  if (allocate_status/=0) then
    print *, &
      'Allocate failed. Program terminates'
    stop 20
  end if
  print 100, heading2(1), time_difference()
  y_i32 = int(x_sp*1000000000, i32)
  deallocate (x_sp)
  print 100, heading2(2), time_difference()
  call sort_data(y_i32, n)
  print 100, heading2(3), time_difference()
  write (unit=100, fmt='(a)') &
    'First 10 32 bit integers'
  write (unit=100, fmt=120) y_i32(1:10)
120 format (5(2x,i10))
  deallocate (y_i32)

  print *, heading1(3)
```

```
allocate (x_dp(1:n), stat=allocate_status)
if (allocate_status/=0) then
  print *, &
    'Allocate failed. Program terminates'
  stop 30
end if


print 100, heading2(1), time_difference()
call random_number(x_dp)
print 100, heading2(2), time_difference()
call sort_data(x_dp, n)
print 100, heading2(3), time_difference()
write (unit=100, fmt='(a)') &
  'First 10 64 bit reals'
write (unit=100, fmt=110) x_dp(1:10)

print *, heading1(4)

allocate (y_i64(1:n), stat=allocate_status)
if (allocate_status/=0) then
  print *, &
    'Allocate failed. Program terminates'
  stop 40
end if

print 100, heading2(1), time_difference()
y_i64 = int(x_dp*1000000000000000_i64, i64)
deallocate (x_dp)
print 100, heading2(2), time_difference()
call sort_data(y_i64, n)
print 100, heading2(3), time_difference()
write (unit=100, fmt='(a)') &
  'First 10 64 bit integers'
write (unit=100, fmt=120) y_i64(1:10)
deallocate (y_i64)

print *, heading1(5)

allocate (x_qp(1:n), stat=allocate_status)
if (allocate_status/=0) then
  print *, &
    'Allocate failed. Program terminates'
  stop 50
end if
```

```
    print 100, heading2(1), time_difference()
    call random_number(x_qp)
    print 100, heading2(2), time_difference()
    call sort_data(x_qp, n)
    print 100, heading2(3), time_difference()
    write (unit=100, fmt='(a)') &
      'First 10 128 bitreals'
    write (unit=100, fmt=110) x_qp(1:10)

    close (200)
    print *, 'Program terminates'
    call end_timing()

end program ch3801
```

Table 38.1 has timing information for four compilers.

Table 38.1 Generic recursive quicksort timing

		gfortran	Intel	Nag	Oracle	Mean	StdDev
Allocate	32 bit real	0.008	0.000	0.000	0.000	0.002	0.004
Allocate	32 bit int	0.000	0.000	0.000	0.008	0.002	0.004
Allocate	64 bit real	0.094	0.031	0.031	0.000	0.039	0.039
Allocate	64 bit int	0.016	0.000	0.016	0.000	0.008	0.009
Allocate	128 bit real	0.156	0.047	0.047	0.000	0.063	0.066
Allocate	Total	0.274	0.078	0.094	0.008	0.114	0.113
Random	32 bit real	0.562	0.422	0.609	2.125	0.930	0.801
Random	32 bit int	0.219	0.172	0.328	0.062	0.195	0.110
Random	64 bit real	1.492	0.594	0.531	2.219	1.209	0.804
Random	64 bit int	0.414	0.328	0.609	0.133	0.371	0.197
Random	128 bit real	11.203	3.797	1.070	3.625	4.924	4.368
Random	Total	13.890	5.313	3.147	8.164	7.629	4.653
Sort	32 bit real	13.742	12.328	15.063	11.586	13.180	1.541
Sort	32 bit int	3.492	2.891	4.781	2.203	3.342	1.095
Sort	64 bit real	14.945	13.266	16.078	12.664	14.238	1.561
Sort	64 bit int	2.742	2.312	2.906	1.633	2.398	0.568
Sort	128 bit real	45.703	33.141	18.750	36.633	33.557	11.201
Sort	Total	80.624	63.938	57.578	64.719	66.715	9.809
Overall	Total	94.788	69.329	60.819	72.891	74.457	14.469

Here are some simple observations about the timing information in this table:

- allocation is a negligible component of the overall time

- random number generation takes between 5 and 15% of total timing
- integer sorting is much faster than real sorting
- sorting of 32 and 64 bit reals is similar
- overall processing of the Nag format 128 bit real is faster than the other 128 bit formats

38.2 Example 2: Non Recursive Quicksort Example with Timing Details

The subroutines in this example are taken from the Netlib site. They are 3 non recursive Fortran 77 implementation of Quicksort.

Visit the Netlib site for more details.

```
http://www.netlib.org/
```

The following is taken directly from their FAQ.

- What is Netlib? The Netlib repository contains freely available software, documents, and databases of interest to the numerical, scientific computing, and other communities. The repository is maintained by AT&T Bell Laboratories, the University of Tennessee and Oak Ridge National Laboratory, and by colleagues worldwide. The collection is replicated at several sites around the world, automatically synchronized, to provide reliable and network efficient service to the global community.

The routines we are interested in are in the following directory.

```
http://www.netlib.org/slatec/src/
```

Three versions are provided.

```
http://www.netlib.org/slatec/src/isort.f
http://www.netlib.org/slatec/src/ssort.f
http://www.netlib.org/slatec/src/dsort.f
```

They are fixed form Fortran 77. A small set of changes need to be made to enable them to be compiled and used in this example.

We will cover the changes we have made for the double precision sort routine dsort.f.

Here is the subroutine header for the double precision subroutine.

```
SUBROUTINE DSORT (DX, DY, N, KFLAG)
```

The routine takes 4 parameters and we look at the implementation of the `dsort` routine to find out more details about each parameter. This line

```
C***TYPE DOUBLE PRECISION (SSORT-S, DSORT-D, ISORT-I)
```

provides the first clue as to the nature of the parameters.
The following provide some more.

```
C   Description of Parameters
C      DX - array of values to be sorted   (usually abscissas)
C      DY - array to be (optionally) carried along
C      N  - number of values in array DX to be sorted
C      KFLAG - control parameter
C            =  2  means sort DX in increasing order and carry DY
along.
C            =  1  means sort DX in increasing order (ignoring DY)
C            = -1  means sort DX in decreasing order (ignoring DY)
C            = -2  means sort DX in decreasing order and carry DY
along.
```

The following lines then complete the information.

```
C      .. Scalar Arguments ..
       INTEGER KFLAG, N
C      .. Array Arguments ..
       DOUBLE PRECISION DX(*), DY(*)
```

If we set the fourth parameter to 1, we can use the same array for the first and second arguments.

We have made source code changes to the three subroutines.

The changes are summarised below, and we have included details of the line numbers in each sort subroutine. The changes involve commenting out 4 sets of lines.

	Line number(s) in subroutines		
	dsort.f	ssort.f	isort.f
*DECK	1	1	1
EXTERNAL XERMSG	61	60	60
1st Call to XERMSG	66-70	65-69	65-69
2nd call to XERMSG	73-78	72-77	72-77

Here are the lines that need to be commented out.

```
*DECK DSORT
..
..
      EXTERNAL XERMSG
..
..
      IF (NN .LT. 1) THEN
         CALL XERMSG ('SLATEC', 'DSORT',
     +       'The number of values to be sorted is not positive.',
1, 1)
         RETURN
      ENDIF
..
..
      IF (KK.NE.1 .AND. KK.NE.2) THEN
         CALL XERMSG ('SLATEC', 'DSORT',
     +       'The sort control parameter, K, is not 2, 1, -1, or -2.',
2,
     +       1)
         RETURN
      ENDIF
..
..
```

The following lines

```
C***REFERENCES  R. C. Singleton, Algorithm 347, An efficient algorithm
C                 for sorting with minimal storage, Communications
of
C                 the ACM, 12, 3 (1969), pp. 185-187.
C***ROUTINES CALLED  XERMSG
C***REVISION HISTORY  (YYMMDD)
C   761101  DATE WRITTEN
C   761118  Modified to use the Singleton quicksort algorithm.  (JAW)
C   890531  Changed all specific intrinsics to generic.  (WRB)
C   890831  Modified array declarations.  (WRB)
C   891009  Removed unreferenced statement labels.  (WRB)
C   891024  Changed category.  (WRB)
C   891024  REVISION DATE from Version 3.2
C   891214  Prologue converted to Version 4.0 format.  (BAB)
C   900315  CALLs to XERROR changed to CALLs to XERMSG.  (THJ)
C   901012  Declared all variables; changed X,Y to DX,DY; changed
C           code to parallel SSORT. (M. McClain)
C   920501  Reformatted the REFERENCES section.  (DWL, WRB)
C   920519  Clarified error messages.  (DWL)
C   920801  Declarations section rebuilt and code restructured to use
C           IF-THEN-ELSE-ENDIF.  (RWC, WRB)
```

provide details about the algorithm and its revision history. This information is extremely useful when working with the subroutine.

We are now going to look at one solution to the problem of how to integrate the original program and the three sorting subroutines.

The simplest solution is to independently compile the three routines as Fortran 77 source. Here is the Nag compiler command to achieve this

```
nagfor -c -O4 dsort.f ssort.f isort.f
```

On the Windows platform this will generate the following files

```
dsort.o
ssort.o
isort.o
```

The following command will then compile the modern Fortran code and link the Fortran 77 compiled code into the executable.

```
nagfor -O4 ch3802.f90 dsort.o ssort.o isort.o
```

Here is the command line for the Intel compiler to compile the Fortran 77 netlib routines.

```
ifort /c /fast /Qparallel dsort.f ssort.f isort.f
```

Here is the command line for gfortran to compile the Fortran 77 Netlib routines.

```
gfortran -c -O3 -ffast-math -funroll-loops
  dsort.f ssort.f isort.f
```

Here is the main program.

```
include 'precision_module.f90'
include 'integer_kind_module.f90'
include 'timing_module.f90'

program ch3802
  use precision_module
  use integer_kind_module
  use timing_module
  implicit none
  integer, parameter :: n = 100000000
  character *12 :: nn = '100,000,000'
  character *80 :: report_file_name = 'ch3502.report'
  real (sp), allocatable, dimension (:) :: x_sp
  real (dp), allocatable, dimension (:) :: x_dp
  integer (i32), allocatable, dimension (:) :: y_i32

  integer :: allocate_status
  character *20, dimension (5) :: heading1 = &
```

```
    [ '  32 bit real  ', &
    '  32 bit int   ', &
  '  64 bit real  ', &
  '  64 bit int   ', &
       ' 128 bit real  ' ]

    character *20, dimension (3) :: heading2 = &
    [ '      Allocate  ', &
    '      Random    ', &
  '      Sort      ' ]

    allocate_status = 0

    print *, 'Program starts'
    print *, 'N = ', nn
    call start_timing()

    open (unit=100, file=report_file_name)

    print *, heading1(1)

    allocate (x_sp(1:n), stat=allocate_status)
    if (allocate_status/=0) then
      print *, ' Allocate failed. Program terminates'
      stop 10
    end if
    print 100, heading2(1), time_difference()
  100 format (a20, 2x, f8.3)
    call random_number(x_sp)
    print 100, heading2(2), time_difference()
    call ssort(x_sp, x_sp, n, 1)
    print 100, heading2(3), time_difference()
    write (unit=100, fmt='(a)') &
    ' First 10 32 bit reals'
    write (unit=100, fmt=110) x_sp(1:10)
  110 format (5(2x,e14.6))

    print *, heading1(2)

    allocate (y_i32(1:n), stat=allocate_status)
    if (allocate_status/=0) then
      print *, ' Allocate failed. Program terminates'
      stop 20
    end if
    print 100, heading2(1), time_difference()
```

```
    y_i32 = int(x_sp*1000000000, i32)
    deallocate (x_sp)
    print 100, heading2(2), time_difference()
    call isort(y_i32, y_i32, n, 1)
    print 100, heading2(3), time_difference()
    write (unit=100, fmt='(a)') &
    'First 10 32 bit integers'
    write (unit=100, fmt=120) y_i32(1:10)
  120 format (5(2x,i10))
    deallocate (y_i32)


    print *, heading1(3)

    allocate (x_dp(1:n), stat=allocate_status)
    if (allocate_status/=0) then
      print *, ' Allocate failed. Program terminates'
      stop 30
    end if
    print 100, heading2(1), time_difference()
    call random_number(x_dp)
    print 100, heading2(2), time_difference()
    call dsort(x_dp, x_dp, n, 1)
    print 100, heading2(3), time_difference()
    write (unit=100, fmt='(a)') &
    'First 10 64 bit reals'
    write (unit=100, fmt=110) x_dp(1:10)

    print *, ' Program terminates'
    call end_timing()

  end program ch3802
```

It is then possible to link to the already compiled subroutines when compiling the main program.

The following command will then compile the modern Fortran code and link the Fortran 77 compiled code into the executable using the Nag compiler.

```
nagfor -O4 ch3802.f90 dsort.o ssort.o isort.o
```

Table 38.2 summarises the timing information for the above four compilers.

Table 38.2 Non recursive quicksort timing

		gfortran	Intel	Nag	Oracle	Mean	StdDev
Allocate	32 bit real	0.000	0.016	0.008	0.000	0.006	0.008
Allocate	32 bit int	0.000	0.000	0.000	0.000	0.000	0.000
Allocate	64 bit real	0.094	0.023	0.031	0.004	0.038	0.039
Allocate	Sum	0.094	0.039	0.039	0.004	0.044	0.037
Random	32 bit real	0.562	0.609	0.625	2.062	0.965	0.732
Random	32 bit int	0.203	0.375	0.297	0.066	0.235	0.133
Random	64 bit real	1.484	0.523	0.516	2.090	1.153	0.772
Random	Sum	2.249	1.507	1.438	4.218	2.353	1.296
Sort	32 bit real	11.508	11.563	11.852	12.207	11.783	0.321
Sort	32 bit int	2.945	2.961	3.000	2.242	2.787	0.364
Sort	64 bit real	12.625	12.406	12.320	12.953	12.576	0.282
Sort	Sum	27.078	26.930	27.172	27.402	27.146	0.198
Overall	Sum	29.421	28.476	28.649	31.624	29.543	1.447

Here are some simple observations about the timing information in this table:

- allocation is again a negligible component of the overall time
- random number generation takes between 5% and 15% of total timing
- integer sorting is much faster than real sorting
- sorting of 32 and 64 bit reals is similar
- the sums for the sorting are very similar, as the standard deviation shows

38.2.1 Notes - Version Control Systems

The original program had the following statement

```
*DECK DSORT
```

and this statement was one used in version control or revision control software of the time. Two version control programs that were available on CDC systems from the 1970s and 1980s were called `update` and `modify` that used the above. In computer programming, revision control is any practice that tracks and provides control over changes to source code. Software developers also use revision control software to maintain documentation and configuration files as well as source code.

The use of this kind of software is common for medium to large scale program development.

Wikipedia provides a comparison of what is currently available. See

```
http://en.wikipedia.org/wiki/
Comparison_of_revision_control_software
```

for more information.

38.3 Subroutine and Function Libraries

A software library is a set of precompiled program units (functions and subroutines) that has been written to solve common problems.

In a university environment many departments (e.g. Mechanical Engineering, Electrical Engineering, Mathematics, Physics etc) have libraries that solve common problems in each discipline.

38.4 The Nag Library for SMP and Multicore

The major commercial cross platform numerical library is the Nag library. Nag provide an SMP and multicore version of their library.

More information can be found at:-

```
https://www.nag.co.uk/numeric/numerical_libraries.asp
https://www.nag.co.uk/numeric/FL/FSdescription.asp
```

The library is available on a range of platforms.

- Windows
- Linux (including 64-bit)
- Solaris
- Mac OS X
- AIX

A subset of the library is thread safe.

Many of the algorithms, or routines, in the library are specifically tuned to run significantly faster on multi-socket and multicore systems. We will look at timing information for one of the sorting routines and compare the times to those of our serial sorting routines.

38.5 Example 3: Calling the Nag m01caf Sorting Routine

Here is the program source.

```
include 'precision_module.f90'
include 'timing_module.f90'

program ch3803

  use precision_module
  use timing_module

  implicit none
  integer, parameter :: n = 100000000
  character *12 :: nn = '100,000,000'
  character *80 :: report_file_name = 'ch3505.report'

  real (dp), allocatable, dimension (:) :: x_dp

  integer :: allocate_status = 0
  integer :: ifail = 0

  character *20, dimension (5) :: heading1 = &
    [ '  32 bit real', &
      '  32 bit int ', &
      '  64 bit real', &
      '  64 bit int ', &
      ' 128 bit real' ]

  character *20, dimension (3) :: heading2 = &
    [ '      Allocate ', &
      '      Random   ', &
      '      Sort     ' ]

  print *, 'Program starts'
  print *, 'N = ', nn
  call start_timing()

  open (unit=100, file=report_file_name)

100 format (a20, 2x, f8.3)
110 format (5(2x,e14.6))
120 format (5(2x,i10))

  print *, heading1(3)

  allocate (x_dp(1:n), stat=allocate_status)
  if (allocate_status/=0) then
    print *, 'Allocate failed. Program terminates'
```

```
    stop 30
  end if

  print 100, heading2(1), time_difference()
  call random_number(x_dp)
  print 100, heading2(2), time_difference()
  call m01caf(x_dp, 1, n, 'A', ifail)
  if (ifail/=0) then
    print *, 'sort failed. Program terminates'
    stop 100
  end if

  print 100, heading2(3), time_difference()
  write (unit=100, fmt='(a)') 'First 10 64 bit reals'
  write (unit=100, fmt=110) x_dp(1:10)

  close (200)
  print *, 'Program terminates'
  call end_timing()

end program ch3803
```

Table 38.3 has details of timing information for the serial sorting algorithms.

Table 38.3 Sixty four bit real sort timings

		gfortran	Intel	Nag	Oracle	Mean	StdDev
Recursive sort	64 bit real	14.945	13.266	16.078	12.664	14.238	1.561
Non-recursive Sort	64 bit real	12.625	12.406	12.320	12.953	12.576	0.282

The non recursive solution is faster for three out of four compilers.
Table 38.4 has the timing information for the Nag SMP routine, for 1–8 cores.

Table 38.4 Nag sort m01caf timing

N threads	Time
1	11.938
2	6.773
3	5.047
4	4.211
5	4.094
6	3.703
7	3.586
8	3.391

As can be seen the Nag `m01caf` timing is faster for one core and shows a very impressive speed up as the number of cores goes up. The system is an Intel I7 system, which has 4 physical cores and is also hyper-threaded giving 8 cores overall.

This link

```
https://www.nag.co.uk/numeric/fl/
performance_examples.asp
```

has some examples of how the NAG SMP library performance scales on multiple cores. At the time of writing they were drawn from the following library chapters

- Sorting
- Correlation and Regression Analysis
- Wavelet Transforms
- Interpolation
- Random number generators
- Special Functions

This link

```
https://www.nag.co.uk/numeric/fl/
nagdoc_fl24/html/GENINT/smptuned.html
```

has details of the tuned routines in the SMP library.

Here are some details that were correct at the time of writing.

- There are 77 tuned LAPACK routines
- There are 149 Tuned NAG-specific routines within the Library

The Nag library may well offer you a very cost effective way to improve the speed of your programs. Nag have effectively done the work of parallelising many common problems and sub problems and thus the use of their library routines may save you significant development time and help you produce programs that run faster.

As you are probably aware by now parallelising your own code can be hard work!

38.6 Example 4: Sorting an Array of a Derived Type

In this section we look at rewriting the quicksort algorithm to work with an array of a user defined type, or Fortran derived type. We will use the date data type from an earlier chapter to illustrate the key points.

38.6.1 Compare Function

For each derived type the user needs needs to implement a logical function that compares two variables of that type. This comparison function will replace the $<$ and $>$ comparison tests in the quicksort sorting routine.

38.6.2 Fortran Sources

There are three source files:

- The date module with comparison function
- the new sort routine
- the Fortran test program

They are listed below.

38.6.3 Date Module

```
module date_module

  implicit none

  private

  type, public :: date
    private
    integer :: day
    integer :: month
    integer :: year
  end type date

  character (9) :: day(0:6) = (/ 'Sunday   ', &
    'Monday   ', 'Tuesday  ', 'Wednesday', &
    'Thursday ', 'Friday   ', 'Saturday ' /)
  character (9) :: month(1:12) = (/ 'January  ', &
    'February ', 'March    ', 'April    ', &
    'May      ', 'June     ', 'July     ', &
    'August   ', 'September', 'October  ', &
    'November ', 'December ' /)


  public :: calendar_to_julian, date_, &
    date_to_day_in_year, date_to_weekday_number, &
```

```
    get_day, get_month, get_year, &
    julian_to_date, &
    julian_to_date_and_week_and_day, ndays, &
    print_date, year_and_day_to_date, less_than


contains

  function calendar_to_julian(x) result (ival)
    implicit none
    integer :: ival
    type (date), intent (in) :: x

    ival = x%day - 32075 + 1461*(x%year+4800+(x% &
      month-14)/12)/4 + 367*(x%month-2-((x%month &
      -14)/12)*12)/12 - 3*((x%year+4900+(x%month &
      -14)/12)/100)/4
  end function calendar_to_julian

  function date_(dd, mm, yyyy) result (x)
    implicit none
    type (date) :: x
    integer, intent (in) :: dd, mm, yyyy

    x = date(dd, mm, yyyy)
  end function date_

  function date_to_day_in_year(x)
    implicit none
    integer :: date_to_day_in_year
    type (date), intent (in) :: x
    intrinsic modulo

    date_to_day_in_year = 3055*(x%month+2)/100 - &
      (x%month+10)/13*2 - 91 + (1-(modulo(x%year &
      ,4)+3)/4+(modulo(x%year,100)+99)/100-( &
      modulo(x%year,400)+399)/400)*(x%month+10)/ &
      13 + x%day
  end function date_to_day_in_year

  function date_to_weekday_number(x)
    implicit none
    integer :: date_to_weekday_number
    type (date), intent (in) :: x
    intrinsic modulo
```

```
  date_to_weekday_number = modulo((13*( &
    x%month+10-(x%month+10)/13*12)-1)/5+x%day+ &
    77+5*(x%year+(x%month-14)/12-(x%year+ &
    (x%month-14)/12)/100*100)/4+(x%year+(x% &
    month-14)/12)/400-(x%year+(x%month- &
    14)/12)/100*2, 7)
end function date_to_weekday_number


function get_day(x)
  implicit none
  integer :: get_day
  type (date), intent (in) :: x

  get_day = x%day
end function get_day

function get_month(x)
  implicit none
  integer :: get_month
  type (date), intent (in) :: x

  get_month = x%month
end function get_month

function get_year(x)
  implicit none
  integer :: get_year
  type (date), intent (in) :: x

  get_year = x%year
end function get_year

function julian_to_date(julian) result (x)
  implicit none
  integer, intent (in) :: julian
  integer :: l, n
  type (date) :: x

  l = julian + 68569
  n = 4*l/146097
  l = l - (146097*n+3)/4
  x%year = 4000*(l+1)/1461001
  l = l - 1461*x%year/4 + 31
```

```
  x%month = 80*l/2447
  x%day = l - 2447*x%month/80
  l = x%month/11
  x%month = x%month + 2 - 12*l
  x%year = 100*(n-49) + x%year + 1
end function julian_to_date


subroutine julian_to_date_and_week_and_day(jd, &
  x, wd, ddd)
  implicit none
  integer, intent (out) :: ddd, wd
  integer, intent (in) :: jd
  type (date), intent (out) :: x

  x = julian_to_date(jd)
  wd = date_to_weekday_number(x)
  ddd = date_to_day_in_year(x)
end subroutine julian_to_date_and_week_and_day

logical function less_than(x1, x2)
  implicit none
  type (date), intent (in) :: x1
  type (date), intent (in) :: x2

  if (calendar_to_julian(x1)< &
    calendar_to_julian(x2)) then
    less_than = .true.
  else
    less_than = .false.
  end if
end function less_than

function ndays(date1, date2)
  implicit none
  integer :: ndays
  type (date), intent (in) :: date1, date2

  ndays = calendar_to_julian(date1) - &
    calendar_to_julian(date2)
end function ndays


function print_date(x, day_names, &
  short_month_name, digits)
```

```
implicit none
type (date), intent (in) :: x
logical, optional, intent (in) :: day_names, &
  short_month_name, digits
character (40) :: print_date
integer :: pos
logical :: want_day, want_short_month_name, &
  want_digits
intrinsic len_trim, present, trim

want_day = .false.
want_short_month_name = .false.
want_digits = .false.
print_date = ' '
if (present(day_names)) then
  want_day = day_names
end if
if (present(short_month_name)) then
  want_short_month_name = short_month_name
end if
if (present(digits)) then
  want_digits = digits
end if
if (want_digits) then
  write (print_date(1:2), '(i2)') x%day
  print_date(3:3) = '/'
  write (print_date(4:5), '(i2)') x%month
  print_date(6:6) = '/'
  write (print_date(7:10), '(i4)') x%year
else
  if (want_day) then
    pos = date_to_weekday_number(x)
    print_date = trim(day(pos)) // ' '
    pos = len_trim(print_date) + 2
  else
    pos = 1
    print_date = ' '
  end if
  write (print_date(pos:pos+1), '(i2)') &
    x%day
  if (want_short_month_name) then
    print_date(pos+3:pos+5) = month(x%month) &
      (1:3)
    pos = pos + 7
  else
```

```
        print_date(pos+3:) = month(x%month)
        pos = len_trim(print_date) + 2
      end if
      write (print_date(pos:pos+3), '(i4)') &
        x%year
    end if


    return
  end function print_date

  function year_and_day_to_date(year, day) &
    result (x)
    implicit none
    type (date) :: x
    integer, intent (in) :: day, year
    integer :: t
    intrinsic modulo

    x%year = year
    t = 0
    if (modulo(year,4)==0) then
      t = 1
    end if
    if (modulo(year,400)/=0 .and. &
      modulo(year,100)==0) then
      t = 0
    end if
    x%day = day
    if (day>59+t) then
      x%day = x%day + 2 - t
    end if
    x%month = ((x%day+91)*100)/3055
    x%day = (x%day+91) - (x%month*3055)/100
    x%month = x%month - 2
    if (x%month>=1 .and. x%month<=12) then
      return
    end if
    write (unit=*, fmt='(a,i11,a)') '$$year_and_d&
      &ay_to_date: day of the year input &
      &=', day, ' is out of range.'
  end function year_and_day_to_date

end module date_module
```

38.6.4 Sort Module

```
module generic_sort_module

! use user_module , internal_type => user_type
! less_than is a logical function in the module

  use date_module, internal_type => date

  implicit none

contains

  subroutine sort(x, n)
    integer, intent (in) :: n
    type (internal_type), intent (inout), &
      dimension (n) :: x

    call quicksort(1, n)

  contains

    recursive subroutine quicksort(l, r)
      implicit none
      integer, intent (in) :: l, r
!     local variables
      integer :: i, j
      type (internal_type) :: v, t

      i = l
      j = r
      v = x(int((l+r)/2))
      do
        do while (less_than(x(i),v))
          i = i + 1
        end do
        do while (less_than(v,x(j)))
          j = j - 1
        end do
        if (i<=j) then
          t = x(i)
          x(i) = x(j)
          x(j) = t
          i = i + 1
```

```
            j = j - 1
          end if
          if (i>j) exit
        end do
        if (l<j) then
          call quicksort(l, j)
        end if
        if (i<r) then
          call quicksort(i, r)
        end if
      end subroutine quicksort

    end subroutine sort

  end module generic_sort_module
```

38.6.5 Main Program

```
  include 'ch3804_date_module.f90'
  include 'ch3804_generic_sort_module.f90'
  include 'timing_module.f90'

  program ch3804

    use date_module
    use generic_sort_module
    use timing_module

    implicit none
    integer :: i
    integer, parameter :: n = 1000000
    integer, dimension (1:n) :: julian_dates
    type (date), dimension (n) :: x
    character *20 :: heading

    call start_timing()
    print *, ' '

    open (unit=100, file='julian_dates.dat', &
      form='unformatted')

    heading = 'open'
```

```
  print 100, heading, time_difference()
100 format (a20, f7.3)

  read (100) julian_dates
  heading = 'read'
  print 100, heading, time_difference()

  do i = 1, n
    x(i) = julian_to_date(julian_dates(i))
  end do

  heading = 'copy'
  print 100, heading, time_difference()

  call sort(x, n)

  heading = 'sort'
  print 100, heading, time_difference()
  print *, ' '


  do i = 1, n, 100000
    print *, print_date(x(i))
  end do

  print *, ' '
  call end_timing()

end program ch3804
```

Here is some sample output.

```
 2016/12/ 5 16:56:24 112

open                            0.023
read                            0.004
copy                            0.031
sort                            0.344

  1 January 1859
 31 January 1887
  6 June 1914
  1 November 1941
 28 March 1969
 22 June 1996
```

```
 21 November 2023
 15 March 2051
 20 August 2078
 28 February 2106

 2016/12/ 5 16:56:24 540
```

38.7 Example 5: Binary Search Example

Searching is a common problem in programming. Wirth's book has a short chapter on searching, with coverage of

- linear search
- binary search
- table search
- straight string search
- the Knuth-Morris-Pratt string search
- the Boyer-Moore string search

A linear search of a collection can obviously be quite an expensive operation. The worst case is that the object of interest is the last member of the collection.

In this example we make the assumption that the data is sorted and can then use a very efficient algorithm - a binary search. Here is the program.

```
include 'timing_module.f90'

module character_binary_search_module

contains

  function binary_search(x, n, key) &
    result (position)
    implicit none

!   Algorithm taken from Algorithms +
!   Data Structures - N. Wirth
!   ISBN 0-13-021999-1
!   Pages 57:59
    integer, intent (in) :: n
    character *32, dimension (1:n), &
      intent (in) :: x
    character *32, intent (in) :: key
```

```
    integer :: position
    integer :: l, r, m

    l = 1
    r = n

    do while (l<r)
      m = (l+r)/2
      if (x(m)<key) then
        l = m + 1
      else
        r = m
      end if
    end do

    if (x(r)==key) then
      position = r
    else
      position = 0
    end if

  end function binary_search

end module character_binary_search_module


program ch3805
  use character_binary_search_module
  use timing_module
  implicit none

  integer, parameter :: nwords = 173528
  character *32, dimension (1:nwords) :: &
    dictionary
  character *32 :: word
  character *1 :: answer
  integer :: position

  call start_timing()

  call read_words()

  write (*, 100) time_difference()
100 format (2x, f7.3)
```

```
  do

    print *, &
      'Type in the word you are looking for'
    read *, word

    write (*, 100) time_difference()

    position = binary_search(dictionary, nwords, &
      word)

    write (*, 100) time_difference()

    if (position==0) then
      print *, ' Word not found'
    else
      write (*, 110) trim(word), position
110   format (a, ' found at position ', i6)
    end if

    print *, ' Try again (y/n) ?'
    read *, answer

    if ((answer=='y') .or. (answer=='Y')) then
      cycle
    else
      exit
    end if

  end do

  call end_timing()

contains

  subroutine read_words()
    implicit none
    integer :: i
    character *80 :: file_name = 'words.txt'

    open (unit=10, file=file_name, status='old')
    do i = 1, nwords
      read (10, 100) dictionary(i)
100   format (a)
    end do
```

```
      close (10)

    end subroutine read_words

end program ch3805
```

The program reads in a dictionary. Historically on Unix systems there was a spelling checker, and there would be a *words* file, often in

```
/etc
```

This is an example of one of these files. Many language versions were available. We then search the dictionary to see if the word entered is in the dictionary. The program provides timing information.

Here is the output from a sample run. The data was read from a file.

```
 2015/ 3/10 14:56: 8 430
    0.070
 Type in the word you are looking for
    0.000
    0.000
qwerty found at position 122712
  Try again (y/n) ?
 Type in the word you are looking for
    0.000
    0.000
  Word not found
  Try again (y/n) ?
 Type in the word you are looking for
    0.000
    0.000
albumin found at position   3309
  Try again (y/n) ?
 Type in the word you are looking for
    0.000
    0.000
transubstantiation found at position 158170
  Try again (y/n) ?
 2015/ 3/10 14:56: 8 500
```

As can be seen the timing reading in the file takes less than one tenth of a second, and the search takes less than a microsecond - the resolution made available via the `date_time` subroutine.

The dictionary has over 170,000 words. Handy for Scrabble!

The dictionary word file is called

```
words.txt
```

in the program.

38.8 Problems

38.1 Try out the examples on your system. What timing details do you get?

38.2 Using the non recursive 32 bit integer sort subroutine as a starting point produce a 64 bit integer version. How long did it take to get a working version?

38.3 If you have successfully solved the above problem now produce subroutines for 8 bit and 16 bit integers.

38.4 Using the non recursive 64 bit real subroutine as a starting point produce a 128 bit version. How long did this take?