Sorting

M269 Tutorial Prsntn 2025J

Contents

1 Agenda **2**

2 Adobe Connect **3**
- 2.1 Interface . . . 3
- 2.2 Settings . . . 4
- 2.3 Sharing Screen & Applications . . . 5
- 2.4 Ending a Meeting . . . 5
- 2.5 Invite Attendees . . . 6
- 2.6 Layouts . . . 7
- 2.7 Chat Pods . . . 8
- 2.8 Web Graphics . . . 8
- 2.9 Recordings . . . 8

3 Sorting: Motivation **9**
- 3.1 Sorting as Dances . . . 9
- 3.2 Card Sorting Ex . . . 10

4 Sorting Taxonomy **11**
- 4.1 Sorting Classifications . . . 11

5 Recursion/Iteration **12**

6 Split/Join Sorting **14**
- 6.1 Insertion Sort . . . 15
  - 6.1.1 Insertion Sort — Abstract Algorithm . . . 15
  - 6.1.2 Insertion Sort — Python . . . 15
  - 6.1.3 Activity 2 — Insertion Sort: Trace an Evaluation . . . 16
  - 6.1.4 Insertion Sort — Non-recursive . . . 16
  - 6.1.5 Activity 3 — Insertion Sort Non-recursive Trace . . . 17
- 6.2 Selection Sort . . . 17
  - 6.2.1 Selection Sort — Abstract Algorithm . . . 17
  - 6.2.2 Selection Sort — Python . . . 18
  - 6.2.3 Selection Sort — Non-recursive . . . 18
  - 6.2.4 Activity 5 — Finding the Non-Recursive Algorithm . . . 19
- 6.3 Merge Sort . . . 19
  - 6.3.1 Merge Sort — Abstract Algorithm . . . 19
  - 6.3.2 Merge Sort — Python . . . 20
  - 6.3.3 Merge Sort Diagram . . . 21
  - 6.3.4 Merge Sort Python In-Place . . . 21
- 6.4 Quicksort . . . 23
  - 6.4.1 Quicksort — Abstract Algorithm . . . 23
  - 6.4.2 List Comprehensions . . . 24
  - 6.4.3 Quicksort — Python . . . 24
  - 6.4.4 Quicksort Python In-Place . . . 25

6.5 Bubble Sort . . . 26
6.5.1 Bubble Sort — Abstract Algorithm . . . 26
6.5.2 Bubble Sort — Python . . . 26

7 Non-Comparison Sorts **26**
7.1 Bucket Sort . . . 28
7.2 Radix Sort in Python . . . 28
7.3 Bucket Sort — Development . . . 30
7.4 Radix Sort . . . 32

8 Future Work **33**

9 References **34**
9.1 Sorting Web Links . . . 34
9.2 Python Web Links & References . . . 35
9.3 Haskell Web Links & References . . . 35
9.4 Demonstration 2 Sorting Algorithms as Dances . . . 35
References . . . 36

Python Code Index **39**

Diagrams Index **40**

1 M269 Tutorial Agenda — Sorting, Recursion

- Welcome & introductions
- *Tutorial topics:* Sorting Algorithms, Recursion
- *Adobe Connect* — if you or I get cut off, wait till we reconnect (or send you an email)
- *Time:* about 1.5 hours
- Do ask questions or raise points.
- *Source:* of slides, notes, programs and playing cards:

  M269Tutorial20260104SortingPrsntn2025J/

  www.pmolyneux.co.uk/OU/M269FolderSync/M269TutorialNotes/M269Tutorial20260104SortingPrsntn2025J/

Introductions — Phil

- *Name* Phil Molyneux
- *Background*
  - Undergraduate: Physics and Maths (Sussex)
  - Postgraduate: Physics (Sussex), Operational Research (Brunel), Computer Science (University College, London)
  - Worked in Operational Research, Business IT, Web technologies, Functional Programming
- *First programming languages* Fortran, BASIC, Pascal
- *Favourite Software*

  - Haskell — pure functional programming language
  - Text editors TextMate, Sublime Text — previously Emacs
  - Word processing in LaTeX — all these slides and notes
  - Mac OS X
- *Learning style* — I read the manual before using the software

Introductions — You

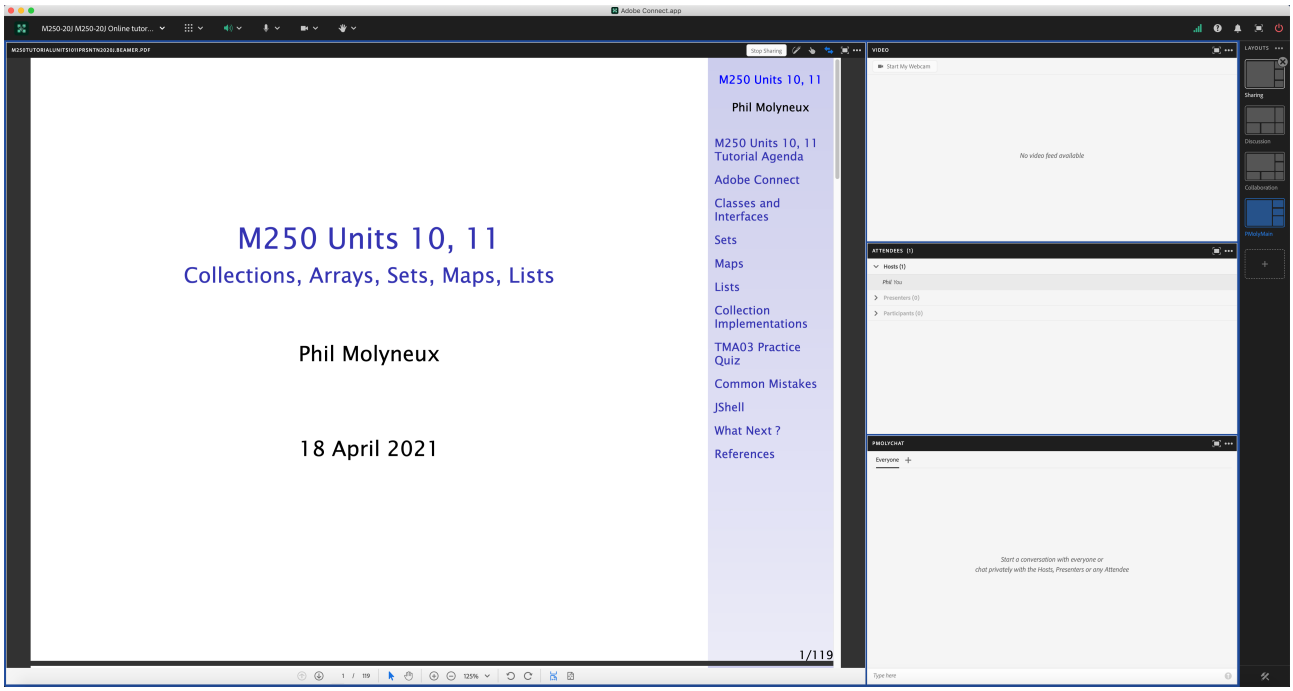
- *Name* ?
- *Favourite software/Programming language* ?
- *Favourite text editor or integrated development environment (IDE)*
- List of text editors, Comparison of text editors and Comparison of integrated development environments
- *Other OU courses* ?
- *Anything else* ?

Go to Table of Contents

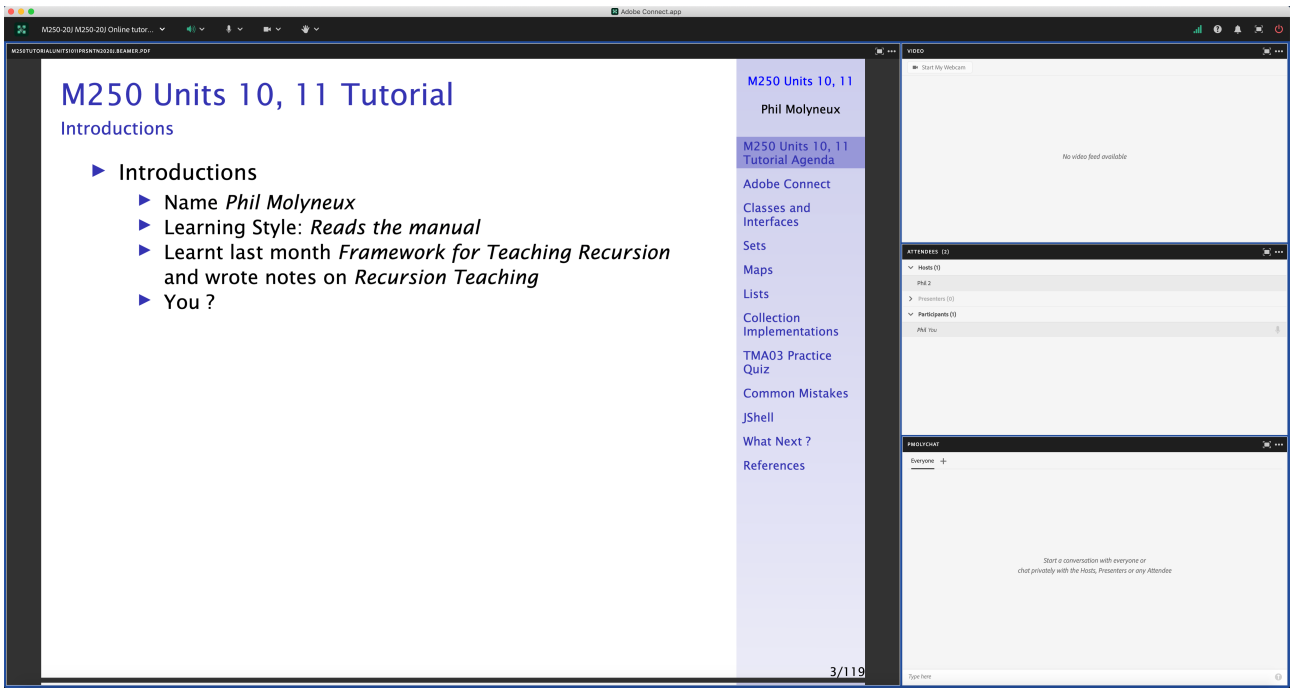
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface

Adobe Connect Interface — Host View



Adobe Connect Interface — Participant View



2.2 Adobe Connect Settings

Adobe Connect — Settings

- **Everybody** *Menu bar* > *Meeting* > Speaker & Microphone Setup
- *Menu bar* > *Microphone* > Allow Participants to Use Microphone ✔
- Check Participants see the entire slide including slide numbers bottom right **Workaround**
  - *Disable Draw* *Share pod* > *Menu bar* > *Draw icon*
  - *Fit Width* *Share pod* > *Bottom bar* > *Fit Width icon* ✔
- *Meeting* > Preferences > General > **Host Cursor** > Show to all attendees
- *Menu bar* > *Video* > Enable **Webcam** for Participants ✔
- Do not *Enable single speaker mode*
- Cancel hand tool
- Do not enable green pointer
- **Recording** Meeting > Record Session ✔
- **Documents** Upload PDF with drag and drop to share pod
- Delete *Meeting* > Manage Meeting Information > Uploaded Content and *check filename* > *click on delete*

Adobe Connect — Access

- **Tutor Access**

  TutorHome > M269 Website > Tutorials

  Cluster Tutorials > M269 Online tutorial room

  Tutor Groups > M269 Online tutor group room

Module-wide Tutorials > M269 Online module-wide room

- **Attendance**

  TutorHome > Students > View your tutorial timetables

- **Beamer Slide Scaling** 440% (422 x 563 mm)
- **Clear Everyone's Status**

  Attendee Pod > Menu > Clear Everyone's Status

- **Grant Access** and send link via email

  Meeting > Manage Access & Entry > Invite Participants...

- **Presenter Only Area**

  Meeting > Enable/Disable Presenter Only Area

Adobe Connect — Keystroke Shortcuts

- Keyboard shortcuts in Adobe Connect
- **Toggle Mic** ⌘+M (Mac), Ctrl+M (Win) (On/Disconnect)
- **Toggle Raise-Hand status** ⌘+E
- **Close dialog box** ⎋ (Mac), Esc (Win)
- **End meeting** ⌘+\

2.3 Adobe Connect — Sharing Screen & Applications

- Share My Screen > Application tab > Terminal for Terminal
- Share menu > Change View > Zoom in for mismatch of screen size/resolution (Participants)
- (Presenter) Change to 75% and back to 100% (solves participants with smaller screen image overlap)
- Leave the application on the original display
- Beware blued hatched rectangles — from other (hidden) windows or contextual menus
- Presenter screen pointer affects viewer display — beware of moving the pointer away from the application
- First time: System Preferences > Security & Privacy > Privacy > Accessibility

2.4 Adobe Connect — Ending a Meeting

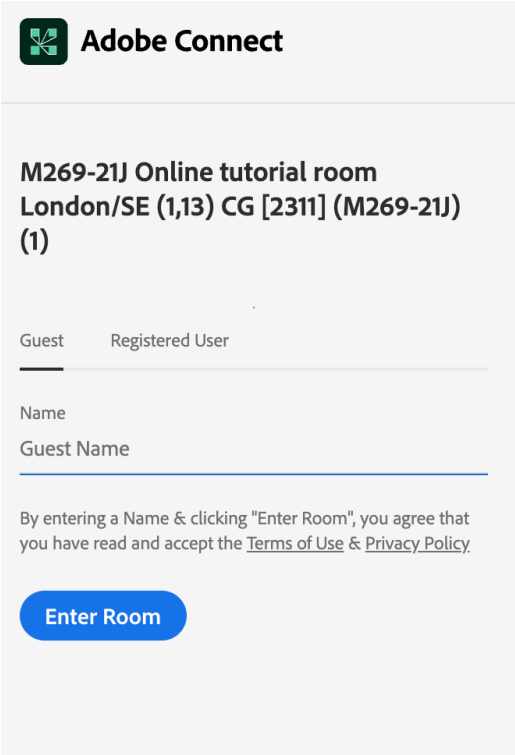
- *Notes for the tutor only*
- **Student:** Meeting > Exit Adobe Connect
- **Tutor:**
- **Recording** Meeting > Stop Recording ✔
- **Remove Participants** Meeting > End Meeting... ✔

  - Dialog box allows for message with default message:
  - *The host has ended this meeting. Thank you for attending.*

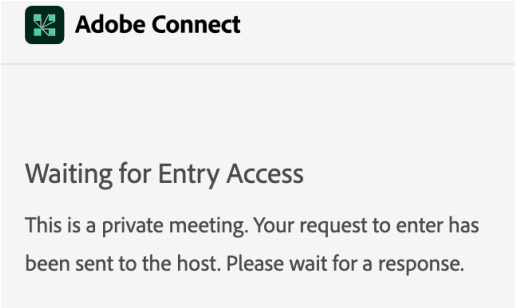
- **Recording availability** *In course Web site for joining the room, click on the eye icon in the list of recordings under your recording* — edit description and name
- **Meeting Information** Meeting > Manage Meeting Information — can access a range of information in Web page.
- **Delete File Upload** Meeting > Manage Meeting Information > Uploaded Content tab select file(s) and click Delete
- **Attendance Report** see course Web site for joining room

2.5 Adobe Connect — Invite Attendees

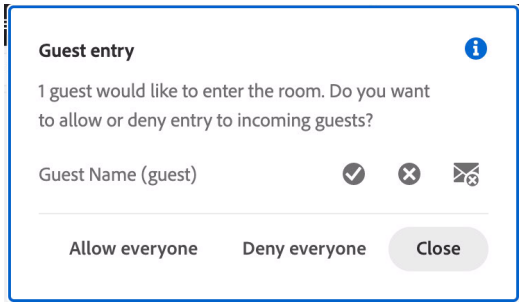
- **Provide Meeting URL** Menu > Meeting > Manage Access & Entry > Invite Participants...
- **Allow Access without Dialog** Menu > Meeting > Manage Meeting Information provides new browser window with *Meeting Information* Tab bar > Edit Information
- Check *Anyone who has the URL for the meeting can enter the room*
- Default *Only registered users and accepted guests may enter the room*
- **Reverts to default next session but URL is fixed**
- Guests have blue icon top, registered participants have yellow icon top — same icon if URL is open
- See Start, attend, and manage Adobe Connect meetings and sessions
- Click on the link sent in email from the Host
- Get the following on a Web page
- As *Guest* enter your name and click on Enter Room



- See the *Waiting for Entry Access* for *Host* to give permission



- *Host* sees the following dialog in *Adobe Connect* and grants access



2.6 Layouts

- **Creating new layouts** example *Sharing* layout
- Menu > Layouts > Create New Layout... > Create a New Layout dialog > Create a new blank layout and name it *PMolyMain*
- New layout has no Pods but does have Layouts Bar open (see Layouts menu)
- **Pods**

- Menu > Pods > Share > Add New Share and resize/position — initial name is *Share n* — rename *PMolyShare*
- **Rename Pod** Menu > Pods > Manage Pods... Manage Pods > Select > Rename or Double-click & rename
- Add Video pod and resize/reposition
- Add Attendance pod and resize/reposition
- Add Chat pod — rename it *PMolyChat* — and resize/reposition
- Dimensions of **Sharing** layout (on 27-inch iMac)
  - Width of Video, Attendees, Chat column 14 cm
  - Height of Video pod 9 cm
  - Height of Attendees pod 12 cm
  - Height of Chat pod 8 cm
- **Duplicating Layouts** does *not* give new instances of the Pods and is probably not a good idea (apart from local use to avoid delay in reloading Pods)
- **Auxiliary Layouts** name *PMolyAux0n*
  - Create new Share pod
  - Use existing Chat pod
  - Use same Video and Attendance pods

2.7 Chat Pods

- **Format Chat text**
- Chat Pod > menu icon > My Chat Color
- Choices: Red, Orange, Green, Brown, Purple, Pink, Blue, Black
- Note: Color reverts to Black if you switch layouts
- Chat Pod > menu icon > Show Timestamps

2.8 Graphics Conversion for Web

- Conversion of the screen snaps for the installation of Anaconda on 1 May 2020
- Using GraphicConverter 11
- File > Convert & Modify > Conversion > Convert
- Select files to convert and destination folder
- Click on Start selected Function or ⌘+↵

2.9 Adobe Connect Recordings

- *Menu bar* > *Meeting* > Preferences > Video
- Aspect ratio > Standard (4:3) (not Wide screen (16:9) default)

- Video quality > Full HD (1080p not High default 480p)
- **Recording** *Menu bar* > *Meeting* > Record Session ✔
- **Export Recording**
- *Menu bar* > *Meeting* > Manage Meeting Information
- *New window* > Recordings > *check Tutorial* > Access Type button
- check Public > check Allow viewers to download
- **Download Recording**
- *New window* > Recordings > *check Tutorial* > Actions > Download File

3 Sorting: Motivation

- Motivation for studying sorting algorithms
- Taxonomy of sorting — see Wikipedia Sorting Algorithm
- Abstract comparison sort — split/join algorithm
- Insertion sort and selection sort described with split/join algorithm diagram and implemented in Python. (A previous edition also included optional Haskell code)
- Recursive and iterative versions
- Mergesort, Quicksort and Bubble sort in the same framework
- Sorting via a data structure — *Tree sort*
- Comparison sorts and Distribution sorts
- Review of Web sites and sorting algorithms used in practice
- From Knuth (1998, page v) *The Art of Computer Programming Vol. 3: Sorting and Searching*
- . . . virtually *every* important aspect of programming arises somewhere in the context of sorting or searching.
- How are good algorithms discovered ?
- How can given algorithms and programs be improved ?
- How can the efficiency of algorithms be analyzed mathematically ?
- How can a person choose rationally between different algorithms for the same task ?
- In what senses can algorithms be proved *best possible* ?
- How does the theory of computing interact with practical considerations ?

3.1 Demonstration 1 Sorting Algorithms as Dances

- AlgoRythmics
- Videos tab > Insertion Sort

- Insertion Sort
- This is the Romanian folk music that inspired Bartók
- Compare the dance with the Python algorithm for Insertion Sort below

ToC

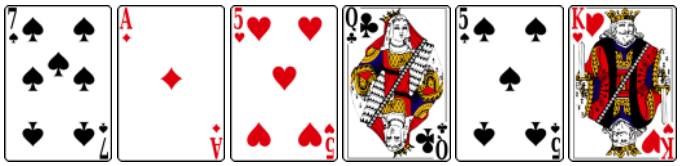
3.2 Activity 1 Card Sorting Exercise

- Almost everyone has played cards and, as part of any card game, will have sorted cards in their hand
- This exercise is aimed at writing down how you sort you cards and giving these instructions to another person to follow.
- Decide on your general ordering of playing cards — you are free to set any ordering you like but here is the usual ordering for suits and values:

```
Clubs < Diamonds < Hearts < Spades

Two < Three < Four < Five < Six
< Seven < Eight < Nine < Ten
< Jack < Queen < King < Ace
```

- Write down your method for sorting cards — the method must specify how to choose a card to move and where to move it to.
- Take the 6 cards given below — record the order of the cards



- Using your method, sort the cards — record the order of the cards after each move of a card
- Now swap your written method and the cards in your original order with another student.
- Follow the other student's method to sort the cards and record your steps

Discussion

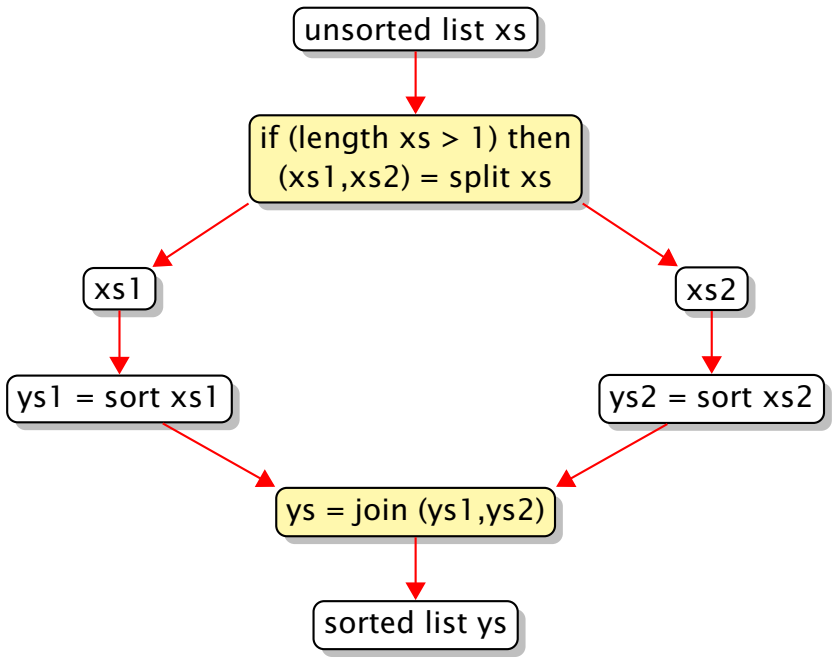
- Did both of you end up with the same sequence of steps?
- Did any of the instructions require human knowledge?
- General point: probably most people use some variation on *Insertion sort* or *Selection sort* but would have steps that had multiple shifts of cards.
- Note: This activity may be done on the *Whiteboard* using cards from http://pmolyneux.co.uk/OU/M269/M269TutorialNotes/M269TutorialSorting/Cards/

ToC

4 Taxonomy of Sorting Algorithms

- Computational complexity — worst, best, average number of comparisons, exchanges and other program contructs (but see http://www.softpanorama.org/Algorithms/sorting.shtml for *Slightly Skeptical View*) — $O(n^2)$ bad, $O(n \log n)$ better
- Other issues: space behaviour, performance on typical data sets, exchanges versus shifts
- Abstract sorting algorithm — Following Merritt (1985); Merritt and Lau (1997) and Azmoodeh (1990, chp 9), we classify the divide and conquer sorting algorithms by easy/hard split/join
- see diagram below

Abstract Sorting Algorithm absSortAlg



4.1 Other Classifications of Sorting Algorithms

- See Wikipedia Sorting algorithm for big list
- *Comparison Sorts*
  - Insertion sort, Selection sort, Merge sort, Quicksort, Bubble sort
  - Sorting via a data structure: Tree sort, Heap sort
- *Non-Comparison sorts* — distribution sorts — bucket sort, radix sort
- *Sorts used in Programming Language Libraries*
  - Powersort and Timsort by Tim Peters — used in Python and Java — combination of merge and insertion sorts
  - Haskell — modified Mergesort by Ian Lynagh in GHC implementation

ToC

5 Recursion and Iteration

- Many functions are naturally defined using recursion
- A recursive function is defined in terms of calls to itself acting on smaller problem instances along with a base case(s) that terminate the recursion
- Classic example: Factorial $n! = n \times (n-1) \cdots 2 \times 1$

```
5 def fac(n) :
6   if n == 1 :
7     return 1
8   else :
9     return n * fac(n-1)
```

- We can evaluate fac(6) by using a substitution model (section 1.1.5) for function application
- *To evaluate a function applied to arguments, evaluate the body of the function with each formal parameter replaced by the corresponding actual arguments.*

  Abelson and Sussman (1996, sec 1.1.5) Structure and Interpretation of Computer Programs

Evaluation of fac(6)

	Expression to Evaluate	Reason
	fac(6)	Initial line 5
→	6 * fac(5)	line 8
→	6 * (5 * fac(4))	line 8
→	6 * (5 * (4 * fac(3))	line 8
→	6 * (5 * (4 * (3 * fac(2))))	line 8
→	6 * (5 * (4 * (3 * (2 * fac(1)))))	line 8
→	6 * (5 * (4 * (3 * (2 * 1))))	line 6
→	720	Arithmetic

- This occupies more space in the process of evaluation since we cannot do the multiplications until we reach the base case of fac()
- This is a *recursive* function and a linear recursive process
- *Implemented* in Python (and most imperative languages) with a *stack* of function calls
- We can define an equivalent *factorial* function that produces a different process

Iterative Factorial

```
24 def facIter(n) :
25   return accProd(n,1)

27 def accProd(n,x) :
28   if n == 1 :
29     return x
30   else :
31     return accProd(n-1, n * x)
```

- facIter() uses accProd() to maintain a running product and accumulate the final result to return

- We can display the evaluation of `facIter(6)` using the substitution model

Evaluation of `facIter(6)`

	Expression to Evaluate	Reason
	`facIter(6)`	Initial line 24
→	`accProd(6,1)`	line 25
→	`accProd(5, 6 * 1)`	line 30 & (*)
→	`accProd(4, 5 * 6)`	line 30 & (*)
→	`accProd(3, 4 * 30)`	line 30 & (*)
→	`accProd(2, 3 * 120)`	line 30 & (*)
→	`accProd(1, 2 * 360)`	line 30 & (*)
→	720	line 28 & (*)

- This occupies constant space — at each stage all the variables describing the state of the calculation are in the function call
- This is a recursive program and an iterative process
- We are assuming the multiplication is evaluated at each function call (strict or eager evaluation)
- Also referred to as tail recursion — we need not build a stack of calls

Iterative Factorial Exercises

- Write a version of the *factorial* function using a `while` loop in Python
- Write a version of the *factorial* function using a `for` loop in Python

Iterative Factorial Exercises — Solutions

- *Factorial* function using a `while` loop in Python

```
46 def facWhile(n) :
47   x = 1

49   while n > 1 :
50     x = n * x
51     n = (n - 1)

53   return x
```

- *Factorial* function using a `for` loop in Python

```
57 def facFor(n) :
58   x = 1

60   for i in range(n,0,-1) :
61     x = i * x

63   return x
```

Tail Recursion and Iteration

- When the structured programming ideas emerged in the 1960s and 1970s the languages such as C and Pascal implemented recursion by always placing the calls on the stack — Python follows this as well

- This means that in those languages they have to have special constructs such as `for` loops, `while` loops, to express iterative processes without recursion
- A `for` loop is syntactically way more complicated than a recursive definition
- Some language implementations (for example, Haskell) spot tail recursion and do not build a stack of calls
- You still have to write your recursion in particular ways to allow the compiler to spot such optimisations.

Structured Programming, GOTO and Recursion

- Böhm and Jacopini (1966) showed that structured programming with a combination of sequence, selection, iteration and procedure calls was Turing complete (see Unit 7)
- In the late 1980s two books came out that were particularly influential:
- Abelson and Sussman (1984, 1996) *Structure and Interpretation of Computer Programs* (known as SICP) which was the programming course for the first year at MIT,
- Bird and Wadler (1988); Bird (1998, 2014) *Introduction to Functional Programming* which was the the programming course for the first year at Oxford.
- See SICP online and Section 1.2 Procedures and the Process They Generate
- Dijkstra (1968) *Go To Statement Considered Harmful* illustrates a debate on structured programming
- The von Neumann computer architecture takes the memory and state view of computation as in Turing m/c
- Lambda calculus is equivalent in computational power to a Turing machine (Turing showed this in 1930s) but efficient implementations did not arrive until 1980s
- Functional programming in Lisp or APL was slow
- Perlis (1982) *Epigrams on Programming*: [Functional programmers] know the value of everything but the cost on nothing
- Meijer et al. (1991) *Recursion is the GOTO of functional programming*
- Leading to common patterns of higher order functions, `map`, `filter`, `fold` and polymorphic data types

ToC

6 Some Split/Join Sorting Algorithms

- Insertion Sort
- Selection Sort
- Merge Sort
- Quicksort
- Bubble Sort

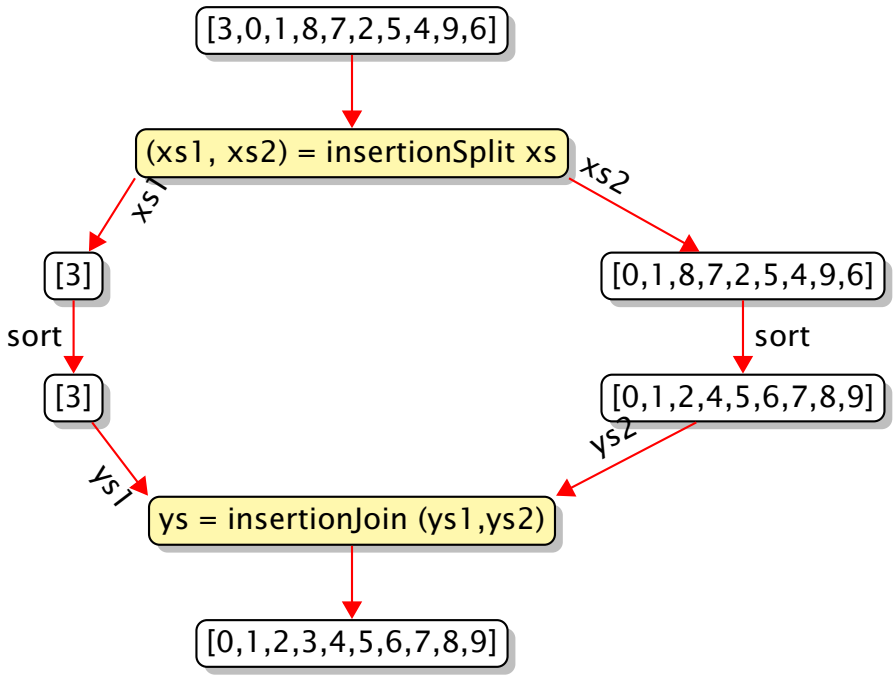
- Implementations in Python, recursive and non-recursive

6.1 Insertion Sort

6.1.1 Insertion Sort — Abstract Algorithm

- *Insertion Split* xs1 is the singleton list of the first item; xs2 is the rest of the list
- *Insertion Join* insert the item in the singleton list into the sorted result of the rest of the list

Abstract Sorting Algorithm Diagram for Insertion Sort insertSortAlg



ToC

6.1.2 Insertion Sort — Python

```
def insSort(xs) :
  if len(xs) <= 1 :
    return xs
  else :
    return ins(xs[0],insSort(xs[1:]))

def ins(x,xs) :
  if xs == [] :
    return [x]
  elif x <= xs[0] :
    return [x] + xs
  else :
    return [xs[0]] + ins(x,xs[1:])
```

Python Rewritten

- In the style of the abstract algorithm

```
def insSort01(xs) :
  if len(xs) <= 1 :
    return xs
  else :
```

```
24      (xs1,xs2) = insertionSplit(xs)
25      ys1 = insSort01(xs1)
26      ys2 = insSort01(xs2)
27      ys = insertionJoin(ys1,ys2)
28      return ys

30  def insertionSplit(xs) :
31    (xs1,xs2) = (xs[0:1],xs[1:])
32    return  (xs1,xs2)

34  def insertionJoin(ys1,ys2) :
35    if ys2 == [] :
36      return ys1
37    elif ys1[0] <= ys2[0] :
38      return ys1 + ys2
39    else :
40      return ys2[0:1] + insertionJoin(ys1,ys2[1:])
```

ToC

6.1.3 Activity 2 — Insertion Sort: Trace an Evaluation

Insertion Sort — Python Recursive

- Evaluation of `insSort([3,0,1,8,7])`

	Expression to Evaluate	Reason
	`insSort([3,0,1,8,7])`	Initial line 4
→	`ins(3, insSort([0,1,8,7]))`	line 7
→	`ins(3, ins(0, insSort([1,8,7])))`	line 7
→	`ins(3, ins(0, ins(1, insSort([8,7]))))`	line 7
→	`ins(3, ins(0, ins(1, ins(8, insSort([7])))))`	line 7
→	`ins(3, ins(0, ins(1, ins(8, [7]))))`	line 5
→	`ins(3, ins(0, ins(1, ([7] + ins(8, [])))))`	line 15
→	`ins(3, ins(0, ins(1, ([7] + [8]))))`	line 11
→	`ins(3, ins(0, ins(1, [7,8])))`	(+) operator
→	`ins(3, ins(0, ([1] + [7,8])))`	line 13
→	`ins(3, ins(0, [1,7,8]))`	(+) operator
→	`ins(3, ([0] + [1,7,8]))`	line 13
→	`ins(3, [0,1,7,8])`	(+) operator
→	`[0] + (ins 3 [1,7,8])`	line 15
→	`[0] + ([1] + (ins 3 [7,8]))`	line 15
→	`[0] + ([1] + ([3] + ([7,8])))`	line 13
→	`[0,1,3,7,8]`	(+) operator

- Note that the evaluation consumes more space in the process of evaluation;
- also note that you need to be careful with the brackets when doing an evaluation like this by hand.

ToC

6.1.4 Insertion Sort — Non-recursive

- The non-recursive version of *Insertion* sort takes each element in turn and inserts it in the ordered list of elements before it.

```
for index = 1 to (len(xs)-1) do
  insert xs[index] in order in xs[0..index-1]
```

- Here is a Python implementation of the above (based on Miller and Ranum (2011, page 215)).
- It uses the *Python Style Guide PEP 8* http://www.python.org/dev/peps/pep-0008/ (Python Enhancement Proposals) — I must admit I prefer indenting with 2 spaces but I imagine the M269 module will have its own guidelines.

```
def insertionSort(xs) :
  for index in range(1, len(xs)) :
    currentValue = xs[index]
    position = index
    while (position > 0) and xs[position - 1] > currentValue :
      xs[position] = xs[position - 1]
      position = position - 1

    xs[position] = currentValue
```

ToC

6.1.5 Activity 3 — Insertion Sort Non-recursive Trace

Insertion Sort — Python Non-recursive

- Evaluation of `insertionSort([3,0,1,8,7])`
- Showing just the outer `for index` loop
3	0	1	8	7	start array

3	0	1	8	7	index = 1
0	3	1	8	7	index = 2
0	1	3	8	7	index = 3
0	1	3	8	7	index = 4
0	1	3	7	8	end

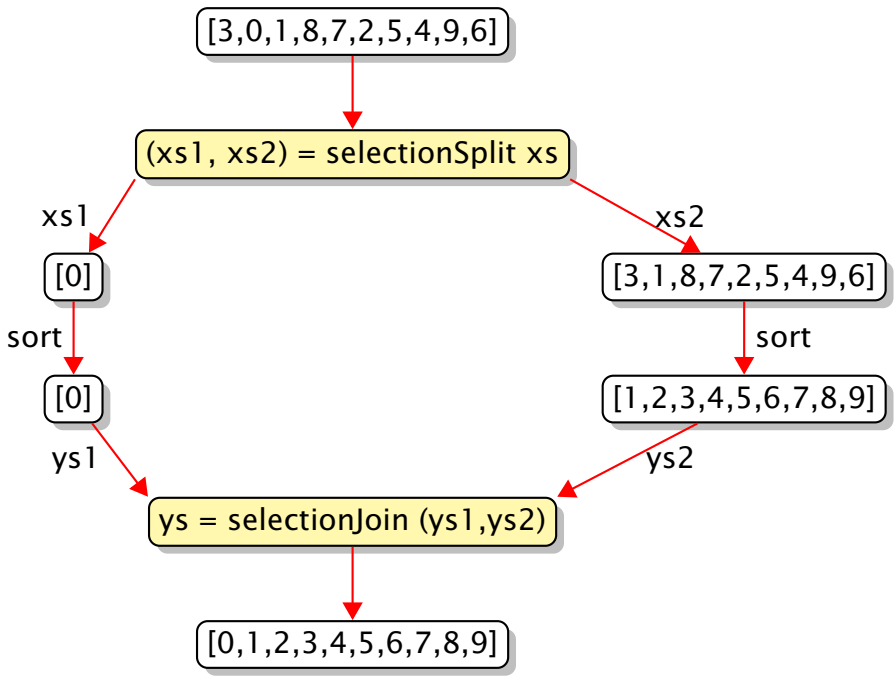
ToC

6.2 Selection Sort

6.2.1 Selection Sort — Abstract Algorithm

- *Selection Split* `xs1` is the singleton list of the minimum item; `xs2` is the original list with the minimum item taken out
- *Selection Join* just put the minimum item and the sorted `xs2` together as the output list

Abstract Sorting Algorithm Diagram for Selection Sort selectSortAlg



ToC

6.2.2 Selection Sort — Python

```python
def selSort(xs) :
  if len(xs) <= 1 :
    return xs
  else :
    minElmnt = min(xs)
    minIndex = xs.index(minElmnt)
    xsWithoutMin = xs[:minIndex] + xs[minIndex+1:]
    return [minElmnt] + selSort(xsWithoutMin)
```

- Why do we not use `xs.remove(min(xs))` ?
- *Why do we not use `xs.remove(min(xs))` ?*
- *`remove()` has the side effect of changing the original argument*
- *If we want `selSort()` to be a function, with no side effects then we should use something else*

ToC

6.2.3 Selection Sort — Non-recursive

- The non-recursive version of *Selection* sort takes each position of the list in turn and swaps the element at that position with the minimum element in the rest of the list from that position to the end of the list.

```
for fillSlot = 0 to (len(xs) - 2) do
  find the minimum of
    xs[fillSlot+1]..xs[len(xs) - 1]
  and swap with xs[fillSlot]
```

Selection Sort — Python Non-recursive Implementation

- Here is a Python implementation of the above (based on Miller and Ranum (2011, page 211) but selecting the smallest first not largest, influenced by http://rosettacode.org/wiki/Sorting_algorithms/Selection_sort#PureBasic).
- Note that here we indent by 2 spaces and use the Python idiomatic *simultaneous assignment* to do the swap in line 71

```
63 def selectionSort(xs) :
64   for fillSlot in range(0,len(xs)-1) :
65     minIndex = fillSlot
66     for index in range(fillSlot+1,len(xs)) :
67       if xs[index] < xs[minIndex] :
68         minIndex = index

70     # if fillSlot != minIndex: # only swap if different
71     xs[fillSlot],xs[minIndex] = xs[minIndex],xs[fillSlot]
```

M & R Non-recursive Selection SOrt

- The non-recursive version of *Selection* sort in Miller & Ranum sorts in ascending order but takes each position of the list in turn from the right end and swaps the element at that position with the maximum element in the rest of the list from the beginning of the list to that position. (Miller and Ranum, 2011, page 211)

```
for fillSlot = len(xs) - 1 down to 1 do
  find the maximum of
    xs[0] .. xs[fillSlot]
  and swap with xs[fillSlot]
```

Selection Sort — Python Non-recursive Implementation

- Here is a Python implementation of the above (based on Miller and Ranum (2011, page 211) selecting the largest first.

```
73 def selSortAscByMax(xs) :
74   for fillSlot in range(len(xs) - 1, 0, -1) :
75     maxIndex = 0
76     for index in range(1, fillSlot + 1) :
77       if xs[index] > xs[maxIndex] :
78         maxIndex = index

80     temp = xs[fillSlot]
81     xs[fillSlot] = xs[maxIndex]
82     xs[maxIndex] = temp
```

- Note that both Python non-recursive versions work by side-effect on the input list — they do not return new lists.

ToC

6.2.4 Activity 5 — Finding the Non-Recursive Algorithm

- For *Insertion Sort* and *Selection Sort* discuss how the non-recursive case can be found by considering the recursive case and doing the algorithm in place.

ToC

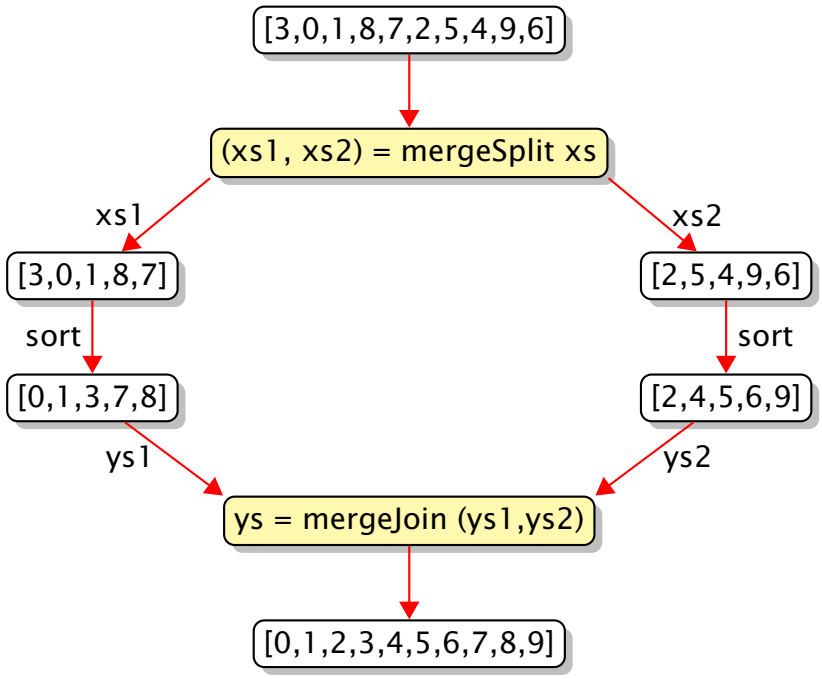
6.3 Merge Sort

6.3.1 Merge Sort — Abstract Algorithm

- *Merge Split* xs1 is half the list; xs2 is the other half of the list.

- *Merge Join* Merge the sorted xs1 and the sorted xs2 together as the output list

Abstract Sorting Algorithm Diagram for Merge Sort mergeSortAlg



ToC

6.3.2 Merge Sort — Python

```python
def mergeSort(xs) :
  if len(xs) <= 1 :
    return xs
  else :
    (aList,bList) = mergeSplit(xs)
    return mergeJoin(mergeSort(aList),mergeSort(bList))

def mergeSplit(xs) :
  return mergeSplit2(xs)

def mergeSplit2(xs) :
  half = len(xs)//2
  return (xs[:half],xs[half:])

def mergeJoin(xs,ys) :
  if xs == [] :
    return ys
  elif ys == [] :
    return xs
  elif xs[0] <= ys[0] :
    return [xs[0]] + mergeJoin(xs[1:],ys)
  else :
    return [ys[0]] + mergeJoin(xs,ys[1:])
```

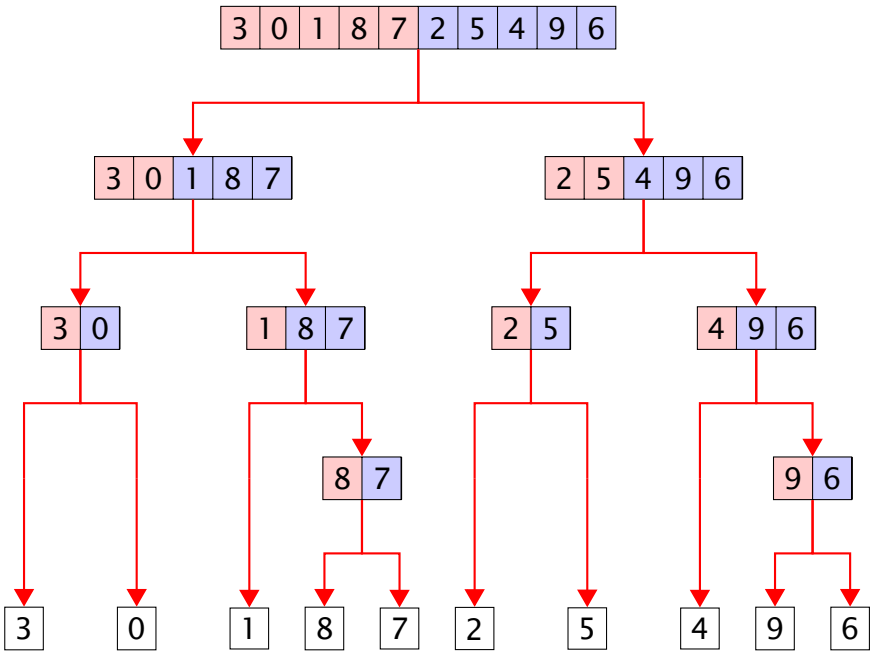
Python mergeSplit1

```python
def mergeSplit1(xs) :
  if len(xs) == 0 :
    return  ([],[])
  elif len(xs) == 1 :
    return (xs,[])
  else :
    (aList,bList) = mergeSplit1(xs[2:])
    return ([xs[0]] + aList, [xs[1]] + bList)
```

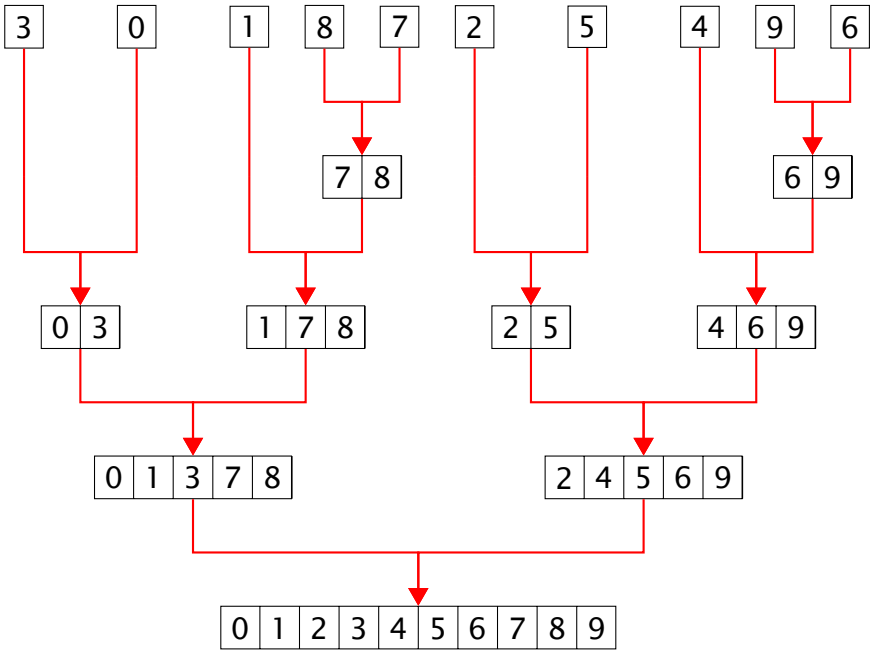
ToC

6.3.3 Merge Sort Diagram

Merge Sort Split Phase mergeSortSplit



Merge Sort Join Phase mergeSortJoin



ToC

6.3.4 Merge Sort Python In-Place

- Here is a Python implementation of the above
- From Miller and Ranum (2011, page 218–221)
- This is also recursive but works in place by changing the array.

- Code from http://interactivepython.org/courselib/static/pythonds/SortSearch/TheMergeSort.html

```
def mergeSortInPlace(xs) :
  if len(xs) > 1 :
    print("Splitting␣", xs)
  else :
    print("Singleton␣", xs)

  if len(xs) > 1 :
    half = len(xs)//2
    (aList, bList) = (xs[:half],xs[half:])

    mergeSortInPlace(aList)
    mergeSortInPlace(bList)
```

```
    i,j,k = 0,0,0
    while i < len(aList) and j < len(bList) :
      if aList[i] < bList[j] :
        xs[k] = aList[i]
        i = i + 1
      else :
        xs[k] = bList[j]
        j = j + 1
      k = k + 1

    while i < len(aList) :
      xs[k] = aList[i]
      i = i + 1
      k = k + 1

    while j < len(bList) :
      xs[k] = bList[j]
      j = j + 1
      k = k + 1
```

- Here is the code that reports the merging of the lists

```
  if len(xs) > 1 :
    print("Merging␣", aList, ",", bList, "to", xs)
  else :
    print("Merged␣", xs)
```

- ␣ is how the listings package shows spaces in strings by default (read the manual)
- // is the Python integer division operator
- aList[start:stop:step] is a *slice* of a list — see Python Sequence Types — slice operations return a new list (van Rossum and Drake, 2011a, page 19) so xs[:] returns a copy (or clone) of xs — if any of the indices are missing or negative than you have to think a bit (or read the manual)
- In Python you really do need to be aware when you are working with values or references to objects.
- A listing of the output of mergeSortInPlace(xsc) below is given in the article version of these notes

```
  >>> from SortingPython import *
  >>> xs = [3,0,1,8,7,2,5,4,9,6]
  >>> xsc = xs[:]
  >>> mergeSortInPlace(xsc)
  Splitting  [3, 0, 1, 8, 7, 2, 5, 4, 9, 6]
  #
  # lines removed
  #
 Merging  [0, 1, 3, 7, 8] , [2, 4, 5, 6, 9]
   to [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Complete listing of output of mergeSortInPlace(xsc)

```
Python3>>> from SortingPython import *
Python3>>> xs = [3,0,1,8,7,2,5,4,9,6]
Python3>>> xsc = xs[:]
Python3>>> mergeSortInPlace(xsc)
Splitting  [3, 0, 1, 8, 7, 2, 5, 4, 9, 6]
Splitting  [3, 0, 1, 8, 7]
Splitting  [3, 0]
Singleton  [3]
Merged  [3]
Singleton  [0]
Merged  [0]
Merging  [3] , [0] to [0, 3]
Splitting  [1, 8, 7]
Singleton  [1]
Merged  [1]
Splitting  [8, 7]
Singleton  [8]
Merged  [8]
Singleton  [7]
Merged  [7]
Merging  [8] , [7] to [7, 8]
Merging  [1] , [7, 8] to [1, 7, 8]
Merging  [0, 3] , [1, 7, 8] to [0, 1, 3, 7, 8]
Splitting  [2, 5, 4, 9, 6]
Splitting  [2, 5]
Singleton  [2]
Merged  [2]
Singleton  [5]
Merged  [5]
Merging  [2] , [5] to [2, 5]
Splitting  [4, 9, 6]
Singleton  [4]
Merged  [4]
Splitting  [9, 6]
Singleton  [9]
Merged  [9]
Singleton  [6]
Merged  [6]
Merging  [9] , [6] to [6, 9]
Merging  [4] , [6, 9] to [4, 6, 9]
Merging  [2, 5] , [4, 6, 9] to [2, 4, 5, 6, 9]
Merging  [0, 1, 3, 7, 8] , [2, 4, 5, 6, 9] to [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

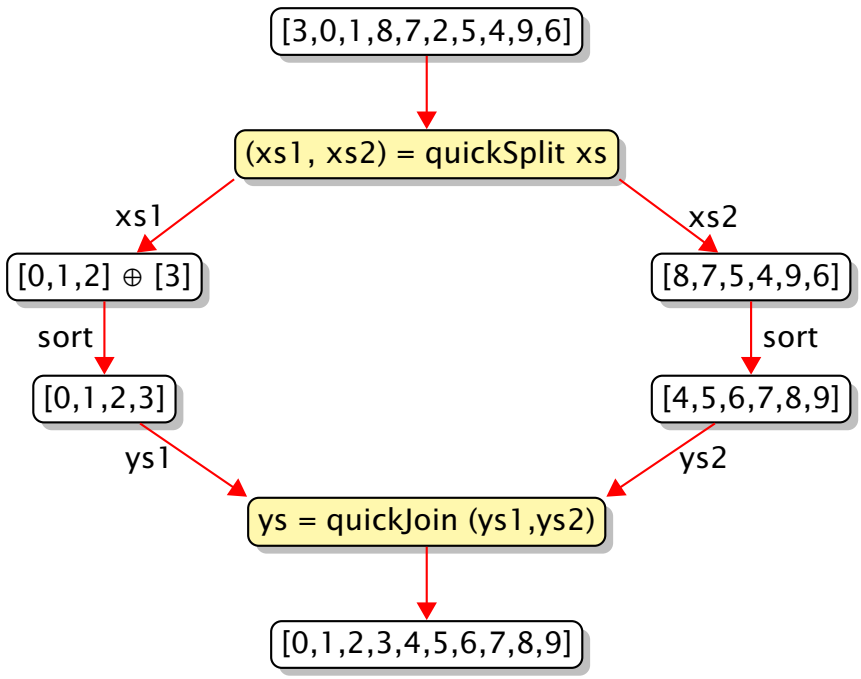
ToC

6.4 Quicksort

6.4.1 Quicksort — Abstract Algorithm

- *Quicksort Split* Choose an item in the list to be the *pivot* item; xs1 comprises items in the list less than the pivot plus the pivot; xs2 comprises items in the list greater than or equal to the pivot.
- *Quicksort Join* just append the sorted xs1 and the sorted xs2 together as the output list

Abstract Sorting Algorithm Diagram for Quicksort quicksortSortAlg



Note: the diagram use ⊕ as the *list append* operator — this is used in various courses and texts

ToC

6.4.2 List Comprehensions

- Haskell 2010 Language Report section 3.11 *List Comprehensions*
- $[e \mid q_1, \ldots, q_n]$, $n \geq 1$ where q_i qualifiers are either
  - *generators* of the form `p <- e` where p is a pattern of type t and e is an expression of type $[t]$
  - *local bindings* that provide new definitions for use in the generated expression e or subsequent boolean guards and generators
  - *boolean guards* which are expressions of type Bool
- Python Language Reference section 6.2.4 *Displays for lists, sets and dictionaries* and section 6.2.5 *List displays*
- `[ expr for target in list]` — simple comprehension
- `[ expr for target in list if condition]` — filters
- `[ expr for target1 in list1 for target2 in list2]` — multiple generators

ToC

6.4.3 Quicksort — Python

```
def qsort(xs) :
  if not xs :
    return []
  else :
    pivot = xs[0]
    less = [x for x in xs     if x <  pivot]
    more = [x for x in xs[1:] if x >= pivot]
    return qsort(less) + [pivot] + qsort(more)
```

- The `if` test at line 160 shows that Python is weakly typed (and the author of this code comes from JavaScript)

ToC

6.4.4 Quicksort Python In-Place

- The in-place version of Quick sort works by partitioning a list in place about a value `pivotvalue`: (Azmoodeh, 1990, page 259–266)

(1) Scan from the left until

  - `alist[leftmark] >= pivotvalue`

(2) Scan from the right until

  - `alist[rightmark] < pivotvalue`

(3) Swap `alist[leftmark]` and `alist[rightmark]`

(4) Repeat (1) to (3) until scans meet

- Here is an in place version of Quick Sort from Miller and Ranum (2011, pages 221–226)
- Code based on http://interactivepython.org/courselib/static/pythonds/SortSearch/TheQuickSort.html

```
168 def quickSort(xs) :
169   quickSortHelper(xs, 0, len(xs) - 1)

171 def quickSortHelper(xs, fst, lst) :
172   if fst < lst :

174     splitPoint = partition(xs,fst,lst)

176     quickSortHelper(xs, fst, splitPoint - 1)
177     quickSortHelper(xs, splitPoint + 1, lst)
```

```
179 def partition(xs,fst,lst) :
180   pivotValue = xs[fst]
181   leftMk     = fst + 1
182   rightMk    = lst
183   done       = False

185   while not done :
186     while leftMk <= rightMk and \
187             xs[leftMk] <= pivotValue :
188       leftMk = leftMk + 1
189     while xs[rightMk] >= pivotValue and \
190             rightMk >= leftMk :
191       rightMk = rightMk - 1

193     if rightMk < leftMk :
194       done = True
195     else :
196       xs[leftMk], xs[rightMk] = xs[rightMk], xs[leftMk]

198   xs[fst], xs[rightMk] = xs[rightMk], xs[fst]
199   return rightMk
```

- The `(\)` is enabling a statement to span multiple lines — see Lutz (2009, page 317), Lutz (2013, page 378)
- for a language that uses the *offside rule* why do we need to do this?

- Note that using `(\)` to create continuations is *frowned on* (Lutz, 2009, page 318), Lutz (2013, page 379)
- the authors should have put the entire boolean expression inside parentheses () so that we get implicit continuation.
- This is not mentioned explicitly in the *Style Guide for Python Code* http://www.python.org/dev/peps/pep-0008/ but it does explicitly mention using Python's implicit line joining with layout guidelines.

ToC

6.5 Bubble Sort

6.5.1 Bubble Sort — Abstract Algorithm

- *Bubble sort* is rather like the *Hello World* program of sorting algorithms — we have to include it even it isn't very useful in practice.
- It can be thought of as an in-place version of Selection sort
- In the implementations below, in each pass through the list, the next highest item is moved (bubbled) to its proper place.
- OK, I should have written it to bubble the smallest the other way to be consistent with the implementations of Selection sort above.

ToC

6.5.2 Bubble Sort — Python

- Here is a Python implementation from Miller and Ranum (2011, pages 207–210)
- it does not test if there have been no swaps but does use some knowledge of the algorithm by reducing the pass length by one each time (which the Haskell one did not do)

```
def bubbleSort(xs) :
  for passNum in range(len(xs) - 1, 0, -1) :
    for i in range(passNum) :
      if xs[i] > xs[i+1] :
        xs[i], xs[i+1] = xs[i+1], xs[i]
```

- Note that `range()` is a built-in function to Python that is used a lot
- Read the documentation at Section 4.6.6 Ranges
- Remember that `range(5)` means `[0,1,2,3,4]` (not `[0,1,2,3,4,5]` or `[1,2,3,4,5]`)

ToC

7 Non-Comparison Sorts

- Sorting without comparing items involves calculating a distribution of items to destinations in the correct order.
- The calculation depends on some structure of an item — for example:
  - digits in a numeral

  - letters in a word

- **Sorting criteria**
- **Speed** Not only $O(f(n))$ or $\Theta(f(n))$ but also the multiplier may be more important — $2n^2$ may be faster than $1000n \log n$ for some n
- **Smooth or Adaptive** the more ordered the input the faster the algorithm
- **Stable** maintain the relative order of records with equal keys
- **Space behaviour** should be compact
- Bird and Gibbons (2020, sec 5.4) *Algorithm Design with Haskell* sec 5.4 Bucketsort and Radixsort
- Wikipedia: Sorting algorithms with links to Pigeonhole sort, Bucket srt, Radix sort
- Knuth (1998, sec 5.2.5) *The Art of Computer Programming Vol. 3: Sorting and Searching* Sec 5.2.5 Sorting by Distribution
- Cormen et al. (2022, chp 8) *Introduction to Algorithms* Chp 8 Sorting in Linear Time
- Sedgewick and Wayne (2011, sec 5.1) *Algorithms* Sec 5.1 String Sorts
- Stubbs and Webre (1989, sec 6.4) *Data Structures with Abstract Data Types and PASCAL* Sec 6.4 Radix Sort
- **Rosetta Code: Sorting algorithms/Radix sort** rosettacode.org/wiki/Sorting_algorithms/Radix_sort
- Erickson (2019) *Algorithms*
- Texts from the 1980s — similar style to many of texts today
- Aho et al. (1974, Sec 3.2) *The Design and Analysis of Computer Algorithms* sec 3.2 Radix sorting
- Aho et al. (1983, sec 8.5) *Data Structures and Algorithms* sec 8.5 Bin Sorting
- Manber (1989, sec 6.4.1) *Introduction to Algorithms: A Creative Approach*
- Reingold and Hansen (1986, sec 8.1.4) *Data Structures in Pascal* sec 8.5 Bin Sorting
- Tilford and Reingold (1986) *Solutions manual to accompany Edward M. Reingold and Wilfred J. Hansen's Data structures in Pascal*
- Baase (1988, sec 2.7) *Computer Algorithms: Introduction to Design and Analysis* sec 2.7 Radix Sorting
- Horowitz and Sahni (1984, sec 7.7) *Fundamentals of Data Structures in Pascal* sec 7.7 Sorting on Several Keys
- **Edsger W Dijkstra 2001** Note on Teaching Programming
  - It is not only the violin that shapes the violinist, we are all shaped by the the tools we train ourselves to use, and in this respect programming languages have a devious influence: they shape our thinking habits.
- The impact of your choice of programming language will be unknown until you encounter a significantly different language

- Early procedural programming languages (C, Pascal, Java, Python) tended to offer arrays as their main data structure or the first to be introduced.
- Early functional programming languages (Liso, SASL) tended to offer lists as the main or first data structure
- Late, procedural languages adopted lists, associative arrays (dictionaries, maps), sets, type annotations, type systems, generics
- Later, functional languages adopted algebraic data types, advanced type systems
- Both evolved libraries and packaging systems

7.1 Bucket Sort

7.2 Radix Sort in Python

- The next two sections develop Radix Sort in Python and Haskell to illustrate different approaches
- They are now meant to be model solutions but try to illustrate the influence of each language
- They both follow Bird and Gibbons (2020, sec 5.4) *Algorithm Design with Haskell* Chp 5 Sorting, section 5.4 Bucketsort and Radixsort
- The *fields* of the items enable the items to be ordered lexicographically
  - digits for numerals
  - letters for words
  - value and suite for playing cards
- It avoids comparison by creating and distributing elements into buckets according to their radix
  - The base for numerals
  - The range in the alphabet for letters in a word
  - Thirteen buckets for values and 4 for suites in playing cards
- The bucketing process is repeated for each field while preserving the order of the prior step
- Sorting from the least significant field provides a stable sort
- Some utility functions

```python
def concat(xss : list[list]) -> [list] :
  """Takes a list of lists and concatenates them together

  """
  ys = [x for xs in xss for x in xs]
  return ys

def pad(k,xs) :
  """ Pad a list with '0' to length k
  """
  cs = str(xs)
  return replicate(k - len(cs),'0') + cs
```

```
def replicate(n,c) :
  """" Returns n copies of the character c

  """"
  def rep(n,c,base) :

    if n == 0 :
      return base
    else :
      return rep(n-1,c,c + base)

  return rep(n,c,'')

def getDigit(n, num) :
    return (str(num)[n])
```

- Two functions to partition list of values into a dictionary of buckets

```
def listToKeyValues(xs, k) :
  """ Partitions a list into buckets
  """

  def lenShow(n) :
    return len(str(n))

  def getDigitPadMaxLen(x) :
    return getDigit(k, pad(maxLen,x))

  maxLen = max(map(lenShow, xs))
  listOfKeyValues = (zip(map(getDigitPadMaxLen,xs),xs))
  return listOfKeyValues

def keyValuesToDict(listKV) :
  kvDict = {str(d): [] for d in range(0,10)}
  for (k,v) in listKV :
    kvDict[k] = kvDict[k] + [v]
  return kvDict
```

```
# Test code

srtTestData01 = [362,745,895,957,54,786,80,543,12,565]

listToDictTest01 \
  = keyValuesToDict(listToKeyValues(srtTestData01, 2))
srtTestData02 = concat(list(listToDictTest01.values()))

listToDictTest02 \
  = keyValuesToDict(listToKeyValues(srtTestData02, 1))
srtTestData03 = concat(list(listToDictTest02.values()))

listToDictTest03 \
  = keyValuesToDict(listToKeyValues(srtTestData03, 0))
srtTestData04 = concat(list(listToDictTest03.values()))
```

```
Python3>>> srtTestData01
[362, 745, 895, 957, 54, 786, 80, 543, 12, 565]
Python3>>> listToDictTest01
{'0': [80], '1': [], '2': [362, 12], '3': [543], '4': [54],
 '5': [745, 895, 565], '6': [786], '7': [957], '8': [], '9': []}
Python3>>> srtTestData02
[80, 362, 12, 543, 54, 745, 895, 565, 786, 957]
Python3>>> listToDictTest02
{'0': [], '1': [12], '2': [], '3': [], '4': [543, 745],
 '5': [54, 957], '6': [362, 565], '7': [],
 '8': [80, 786], '9': [895]}
Python3>>> srtTestData03
[12, 543, 745, 54, 957, 362, 565, 80, 786, 895]
Python3>>> listToDictTest03
{'0': [12, 54, 80], '1': [], '2': [], '3': [362], '4': [],
 '5': [543, 565], '6': [], '7': [745, 786],
 '8': [895], '9': [957]}
Python3>>> srtTestData04
[12, 54, 80, 362, 543, 565, 745, 786, 895, 957]
```

```
def partitionToList(xs,k) :
  listToDictTestOk = keyValuesToDict(listToKeyValues(xs, k))
  srtTestDataOk = concat(list(listToDictTestOk.values()))
  return srtTestDataOk

def radixsort(xs) :

  def lenShow(n) :
    return len(str(n))

  maxLen = max(map(lenShow, xs))

  srtTestData = xs
  for k in range(maxLen - 1 ,-1,-1) :
    srtTestData = partitionToList(srtTestData,k)

  return srtTestData
```

ToC

7.3 Bucket Sort — Development

- This section follows Bird and Gibbons (2020, chp 5) *Algorithm Design with Haskell* Chp 5 Sorting
- In the chapter both the comparison and npn-comparison sorting algorithms are developed around the view of sorting as a two stage process: first build some kind of tree and then flatten it

```
sortD xs = (flatten . mktree) xs
```

- Function application in Haskell is denoted by juxtaposition — so `f x` not `f (x)`
- The `(.)` is the *function composition* operator — `(f . g) x` means `f (g x)`
- Function application is more binding than (almost) anything else and is left associative — so `f x y` means `(f x) y`
- The name `sortD` is used so not to conflict with builtin `sort`
- Items can contain fields which can be used to order the items
- For example, *words* contain *letters*

  Integers contain *digits*
- The fields can be extracted by a list of functions, called *discriminators*, each of which extract one possible field.
- Given a list of discriminators, `ds`, two elements `x` and `y` are lexically ordered if the following returns `True`

```
ordered :: Ord b -> [ a -> b ] -> a -> a -> Bool
ordered [ ] x y = True
ordered (d : ds) x y = (d x < d y)
                       || ((d x == d y) && ordered ds x y)
```

- Initially we divide the words into piles sorted by their first letter
- The piles (or *buckets*) are sortedd the same way but on the next letter and so on
- At the end of this process, there will be a number of little buckets, each containing a single word

- The buckets are then combined in the correct order to give the final sorted list
- We represent the buckets as a tree

```
data TreeD a = LeafD a | NodeD [TreeD a]
```

- This is sometimes called a *rose tree*
- The data structure allows for buckets to have subbuckets
- Building a rose tree

```
mktree :: (Bounded b,Enum b,Ord b) -> [a -> b] -> [a] -> TreeD [a]
mktree [ ] xs = LeafD xs
mktree (d : ds) xs = NodeD (map (mktree ds) (ptn d xs))
```

```
ptn :: (Bounded b,Enum b,Ord b) -> (a -> b) -> [a] -> [[a]]
ptn d xs = [filter (\x -> d x == m) xs | m <- rng]
  where rng = [ minBound . . maxBound ]
```

- The notation `\x -> ...` is a Lambda abstraction see also Lambda Abstractions
- `\x -> x * x` is a function (with no name) which squares its argument
- `(\x -> x * x) 2` evaluates to `2 * 2` and then `4`
- The notation in Bird and Gibbons (2020) *Algorithm Design with Haskell* is

  `λ.x expr`

- This is the notation closer to the orignal Maths notation

```
flatten :: TreeD [a] -> [a]
flatten (LeafD xs) = xs
flatten (NodeD ts) = concatMap flatten ts

bsort ds xs = flatten (mktree ds xs)
```

```
bsort (d : ds) xs
{ definition of bsort }
= flatten (mktree (d : ds) xs)
{ definition of mktree }
= flatten (NodeD (map (mktree ds) (ptn d xs)))
{ definition of flatten }
= concatMap (flatten . mktree ds) (ptn d xs)
{ definition of bsort }
= concatMap (bsort ds) (ptn d xs)
```

```
bsort [] xs   = xs
bsort (d : ds) xs = concatMap (bsort ds) (ptn d xs)
```

- Describe `concatMap`

```
 >>> concatMap (take 3) [[1..], [10..], [100..], [1000..]]
 [1,2,3,10,11,12,100,101,102,1000,1001,1002]
```

```
map (bsort ds) (ptn d xs) = ptn d (bsort ds xs)
```

```
concatMap (bsort ds) (ptn d xs)
{ above stability property }
= concat (ptn d (bsort ds xs))
```

- Here is the definition of `Radixsort`

```
rsort :: (Bounded b,Enum b,Ord b) -> [a -> b] -> [a] -> [a]
rsort [] xs = xs
rsort (d : ds) xs = concat (ptn d (rsort ds xs))
```

```haskell
ptn :: Ix b => (b,b) -> (a -> b) -> [a] -> [[a]]
ptn (l,u) d xs = elems xa
  where xa = accumArray snoc [] (l, u) (zip (map d xs) xs)
  snoc xs x = xs ++ [x]
```

```haskell
ptn (l, u) d xs = map reverse (elems xa)
  where xa = accumArray (flip (:)) [] (l, u) (zip (map d xs) xs)
```

```haskell
rsort :: Ix b => (b,b) -> [a -> b] -> [a] -> [a]
rsort bb [ ] xs = xs
rsort bb (d : ds) xs = concat (ptn bb d (rsort bb ds xs))
```

- Discriminator functions

```haskell
discs :: [Int] ->  [Int -> Char]
discs xs = [\x -> pad k (show x)!!i | i <- [0..k-1]]
   where k = maximum (map (length . show) xs)
pad k xs = replicate (k - length xs) '0' ++ xs
```

```haskell
irsort :: [Int] -> [Int]
irsort xs = rsort ('0', '9') (discs xs) xs
```

ToC

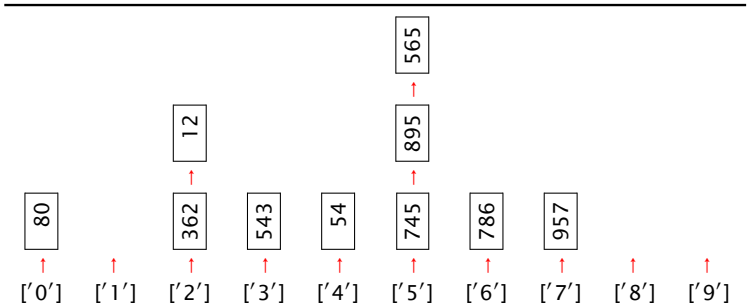
ToC

7.4 Radix Sort

- Radix sort example — sort the following list of integers
- Source Stubbs and Webre (1989, sec 6.4.1) *Data Structures with Abstract Data Types and PASCAL* sec 6.4.1 Example of a Radix Sort

  [362, 745, 895, 957, 054, 786, 080, 543, 012, 565]

- Stage 1 allocate to buckets on the least significant digit

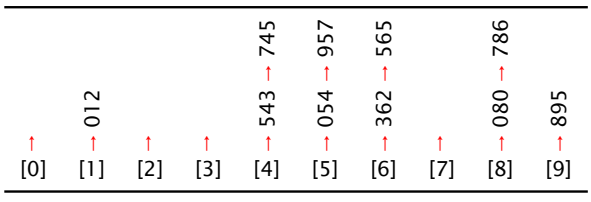


- This results in

  [080,362,012,543,054,745,885,565,786,957]

- Stage 2 allocate to buckets on the second least significant digit

  [080,362,012,543,054,745,885,565,786,957]



- This results in

[012, 543, 745, 054, 957, 362, 565, 080, 885, 786]

- Stage 3 allocate to buckets on the third least significant digit

[012, 543, 745, 054, 957, 362, 565, 080, 786, 895]

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
012 → 054 → 089			362		543 → 565		745 → 786	895	957

- This results in

[012, 054, 080, 362, 543, 565, 745, 786, 895, 957]

- **Radix sort — further example**
- Sorting a pack of cards
- Take a shuffled pack of cards
- Deal the cards to 13 piles for Ace, 2 through 10, Jack, Queen, King
- Pick up the piles *in order*
- Deal to 4 piles for Clubs, Diamonds, Hearts, Spades
- Pick up the piles *in order*
- The pack will now be in order

ToC

ToC

8 Future Work

Programming, Debugging, Psychology

Although programming techniques have improved immensely since the early days, the process of finding and correcting errors in programming — known graphically if inelegantly as *debugging* — still remains a most difficult, confused and unsatisfactory operation. The chief impact of this state of affairs is psychological. Although we are happy to pay lip-service to the adage that to err is human, most of us like to make a small private reservation about our own performance on special occasions when we really try. It is somewhat deflating to be shown publicly and incontrovertibly by a machine that even when we do try, we in fact make just as many mistakes as other people. If your pride cannot recover from this blow, you will never make a programmer.

Christopher Strachey, Scientific American 1966 vol 215 (3) September pp112–124

- To err is human, to really foul things up requires a computer.
- Attributed to Paul R. Ehrlich in 101 Great Programming Quotes
- Attributed to Bill Vaughn in Quote Investigator
- Derived from Alexander Pope (1711, An Essay on Criticism)

- *To Err is Humane; to Forgive, Divine*
- This also contains

  A little learning is a dangerous thing;

  Drink deep, or taste not the Pierian Spring
- In programming, this means you have to *read the fabulous manual* (RTFM)

Sorting, Searching, Binary Trees

- Recursive function definitions
- Inductive data type definitions
  - A *list* is either an empty list or a first item followed by the rest of the list
  - A *binary tree* is either an empty tree or a node with an item and two sub-trees
- Recursive definitions often easier to find than iterative
- Sorting
- Searching
- Both use binary tree structure — either implicitly or explicitly

Future Work — Dates

- Sunday 4 January 2026 Tutorial Online Sorting, Recursion
- Sunday 11 January 2026 Tutorial Online Binary Trees, Recursion
- Sunday 8 February 2026 (Module wide) Tutorial Online Binary Trees, Recursion
- Tuesday, 10 March 2026 TMA02
- Sunday, 8 March 2026 Tutorial (Online): Graphs, Greedy Algorithms
- Sunday, 12 April 2026 Tutorial (Online): (Module wide) Dynamic Programming
- Sunday, 26 April 2026 Tutorial (Online): (Module wide) Computability, Complexity
- Sunday, 3 May 2026 Tutorial (Online): Review of course material for TMA03
- Tuesday, 26 May 2025 TMA03

ToC

9 Web Sites & References

9.1 Sorting Web Links

- **Rosetta Code Sorting Algorithms** http://rosettacode.org/wiki/Sorting_algorithms — sorting algorithms implemented n lots of programming languages
- **Sorting Algorithm Animations** https://www.toptal.com/developers/sorting-algorithms — visual display of the performance of various sorting algorithms for several classes of data: random, nearly sorted, reversed, few unique — worth browsing to.

- **Sorting Algorithms as Dances** https://www.youtube.com/user/AlgoRythmics — inspired!

ToC

9.2 Python Web Links & References

- **Miller and Ranum (2011)** http://interactivepython.org/courselib/static/pythonds/index.html — the entire book online with a nice way of running the code.
- **Lutz (2013)** — one of the best introductory books
- **Lutz (2011)** — a more advanced book — earlier editions of these books are still relevant — you can also obtain electronic versions from the O'Reilly Web site http://oreilly.com
- **Python 3 Documentation** https://docs.python.org/3/
- **Python Style Guide PEP 8** https://www.python.org/dev/peps/pep-0008/ (Python Enhancement Proposals)

ToC

9.3 Haskell Web Links & References

- **Haskell Language** https://www.haskell.org
- **HaskellWiki** https://wiki.haskell.org/Haskell
- **Learn You a Haskell for Great Good!** — very readable introduction to Haskell
- **Real World Haskell** http://book.realworldhaskell.org — more advanced
- **Thompson (2011)** — a good text for functional programming for beginners
- **Bird and Wadler (1988); Bird (1998, 2014)** — one of the best introductions but tough in parts, requires some mathematical maturity — the three books are in effect different editions
- **Bird and Gibbons (2020)** — the descriptions of five main principles of algorithm design: divide and conquer, greedy algorithms, thinning, dynamic programming, and exhaustive search, are mainly language neutral
- **Functors, Applicatives, and Monads in Pictures** — a very good outline with cartoons
- **Typeclassopedia** https://wiki.haskell.org/Typeclassopedia — a more formal introduction to Functors, Applicatives and Monads
- **Haskell Wikibook**

ToC

9.4 Demonstration 2 Sorting Algorithms as Dances

- Quicksort
- https://www.youtube.com/user/AlgoRythmics

- the hats make the point(!)

ToC

References

Abelson, Harold and Gerald Jay Sussman (1984). *Structure and Interpretation of Computer Programs*. MIT Press, first edition. URL http://mitpress.mit.edu/sicp/. 14

Abelson, Harold and Gerald Jay Sussman (1996). *Structure and Interpretation of Computer Programs*. MIT Press, second edition. ISBN 0262510871. URL http://mitpress.mit.edu/sicp/. 12, 14

Aho, Alfred V; John E Hopcroft; and Jeffrey D Ullman (1974). *The Design and Analysis of Computer Algorithms*. Addison Wesley. ISBN 0201000296. 27

Aho, Alfred V; John E Hopcroft; and Jeffrey D Ullman (1983). *Data Structures and Algorithms*. Addison-Wesley. ISBN 0201000237. 27

Azmoodeh, Manoochehr (1990). *Abstract Data Types and Algorithms*. Palgrave Macmillan, second edition. ISBN 0333512103. 11, 25

Baase, Sara (1988). *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley. ISBN 0201060353. 27

Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460. 14, 35

Bird, Richard (2014). *Thinking Functionally with Haskell*. Cambridge University Press. ISBN 1107452643. URL https://www.cs.ox.ac.uk/publications/books/functional/. 14, 35

Bird, Richard and Jeremy Gibbons (2020). *Algorithm Design with Haskell*. Cambridge University Press. ISBN 9781108869041. URL https://www.cs.ox.ac.uk/publications/books/adwh/. 27, 28, 30, 31, 35

Bird, Richard and Phil Wadler (1988). *Introduction to Functional Programming*. Prentice Hall, first edition. ISBN 0134841972. 14, 35

Böhm, Corrado and Giuseppe Jacopini (1966). Flow diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371. 14

Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; and Clifford Stein (2022). *Introduction to Algorithms*. MIT Press, fourth edition. ISBN 9780262046305. URL https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition. 27

Dijkstra, Edsger W (1968). Letters to the editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148. 14

Dromey, R.Geoff (1982). *How to Solve it by Computer*. Prentice-Hall. ISBN 0134340019.

Dromey, R.Geoff (1989). *Program Derivation: The Development of Programs from Specifications*. Addison Wesley. ISBN 0201416247.

Erickson, Jeff (2019). Algorithms. Web (Last Checked 7 February 2021). URL http://jeffe.cs.illinois.edu/teaching/algorithms/. 27

Horowitz, Ellis and Sartaj Sahni (1984). *Fundamentals of Data Structures in Pascal*. Computer Science Press. ISBN 0914894943. 27

Hudak, Paul; John Hughes; Simon Peyton Jones; and Phil Wadler (2007). A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12-1–12-55. ACM New York, NY, USA.

Knuth, D.E. (1998). *The Art of Computer Programming Vol. 3: Sorting and Searching*. The Art of Computer Programming: Sorting and Searching. Adddison Wesley, second edition. ISBN 0201896850. URL https://www-cs-faculty.stanford.edu/~knuth/taocp.html. 9, 27

Lee, Gias Kay (2013). Functional Programming in 5 Minutes. Web. http://gsklee.im, URL http://slid.es/gsklee/functional-programming-in-5-minutes.

Lutz, Mark (2009). *Learning Python*. O'Reilly, fourth edition. ISBN 0596158068. 25, 26

Lutz, Mark (2011). *Programming Python*. O'Reilly, fourth edition. ISBN 0596158106. URL http://learning-python.com/books/about-pp4e.html. 35

Lutz, Mark (2013). *Learning Python*. O'Reilly, fifth edition. ISBN 1449355730. URL http://learning-python.com/books/about-lp5e.html. 25, 26, 35

Manber, Udi (1989). *Introduction to Algorithms: A Creative Approach*. Addison-Wesley. ISBN 0201120372. 27

Marlow, Simon and Simon Peyton Jones (2010). Haskell Language and Library Specification. Web. URL http://www.haskell.org/haskellwiki/Language_and_library_specification.

Meijer, Erik; Maarten Fokkinga; and Ross Paterson (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer. 14

Merritt, SM and KK Lau (1997). A logical inverted taxonomy of sorting algorithms. In *Proceedings of the Twelfth International Symposium on Computer and Information Sciences*, pages 576–583. Citeseer. 11

Merritt, Susan M (1985). An inverted taxonomy of sorting algorithms. *Communications of the ACM*, 28(1):96–99. 11

Miller, Bradley W. and David L. Ranum (2011). *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle Associates Inc, second edition. ISBN 1590282574. URL http://interactivepython.org/courselib/static/pythonds/index.html. 17, 19, 21, 25, 26, 35

Perlis, Alan J. (1982). Epigrams on Programming. *SIGPLAN Notices*, 17(9):7–13. 14

Reingold, Edward M and Wilfred J Hansen (1986). *Data Structures in Pascal*. Little, Brown. ISBN 0316739316. 27

Sedgewick, Robert and Kevin Wayne (2011). *Algorithms*. Addison Wesley, fourth edition. ISBN 032157351X. URL http://algs4.cs.princeton.edu/home/. 27

Stubbs, Daniel and Neil Webre (1989). *Data Structures with Abstract Data Types and PASCAL*. Brooks Cole, second edition. ISBN 0534092640. 27, 32

Sussman, Julie (1985a). *Instructor's Manual to Accompany Structure and Interpretation*

of Computer Programs. MIT Press. ISBN 0262 691019. URL http://mitpress.mit.edu/sites/default/files/sicp/index.html.

Sussman, Julie (1985b). *Instructor's Manual to Accompany Structure and Interpretation of Computer Programs*. MIT Press, second edition. ISBN 0262 692201. URL http://mitpress.mit.edu/sites/default/files/sicp/index.html.

Thompson, Simon (2011). *Haskell the Craft of Functional Programming*. Addison Wesley, third edition. ISBN 0201882957. URL http://www.haskellcraft.com/craft3e/Home.html. 35

Tilford, John S. and Ruth N. Reingold (1986). *Solutions manual to accompany Edward M. Reingold and Wilfred J. Hansen's Data structures in Pascal*. Little, Brown. ISBN 978-0-316-73932-0. 27

van Rossum, Guido and Fred Drake (2011a). *An Introduction to Python*. Network Theory Limited, revised edition. ISBN 1906966133. 22

van Rossum, Guido and Fred Drake (2011b). *The Python Language Reference Manual*. Network Theory Limited, revised edition. ISBN 1906966141.

ToC

Python Code Index

Index for some (but not all) of the python code. Note that the index commands are placed after any *listings* environment containing the code to be indexed.

accProd, 12

bubbleSort, 26

fac, 12
facFor, 13
facIter, 12
facWhile, 13

ins, 15
insertionJoin, 16
insertionSort, 17
insertionSplit, 16
insSort, 15
insSort01, 16

mergeJoin, 20
mergeSort, 20
mergeSortInPlace, 22
mergeSplit, 20
mergeSplit1, 21
mergeSplit2, 20

partition, 25

qsort, 25
quickSort, 25
quickSortHelper, 25

selectionSort, 19
selSort, 18
selSortAscByMax, 19

ToC

Diagrams Index

Index for some of the PGF/TikZ diagrams. Note that the indexing commands are placed after the diagram code

absSortAlg, 11

insertSortAlg, 15

mergeSortAlg, 20

mergeSortJoin, 21

mergeSortSplit, 21

quicksortSortAlg, 24

selectSortAlg, 18

ToC