Sorting

M269 Tutorial

Phil Molyneux

9 January 2022

M269 Tutorial: Sorting, Recursion

Agenda

- 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:
  M269Tutorial03Sorting

  www.pmolyneux.co.uk/OU/M269FolderSync/M269TutorialNotes/M269TutorialSorting

M269 Tutorial

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

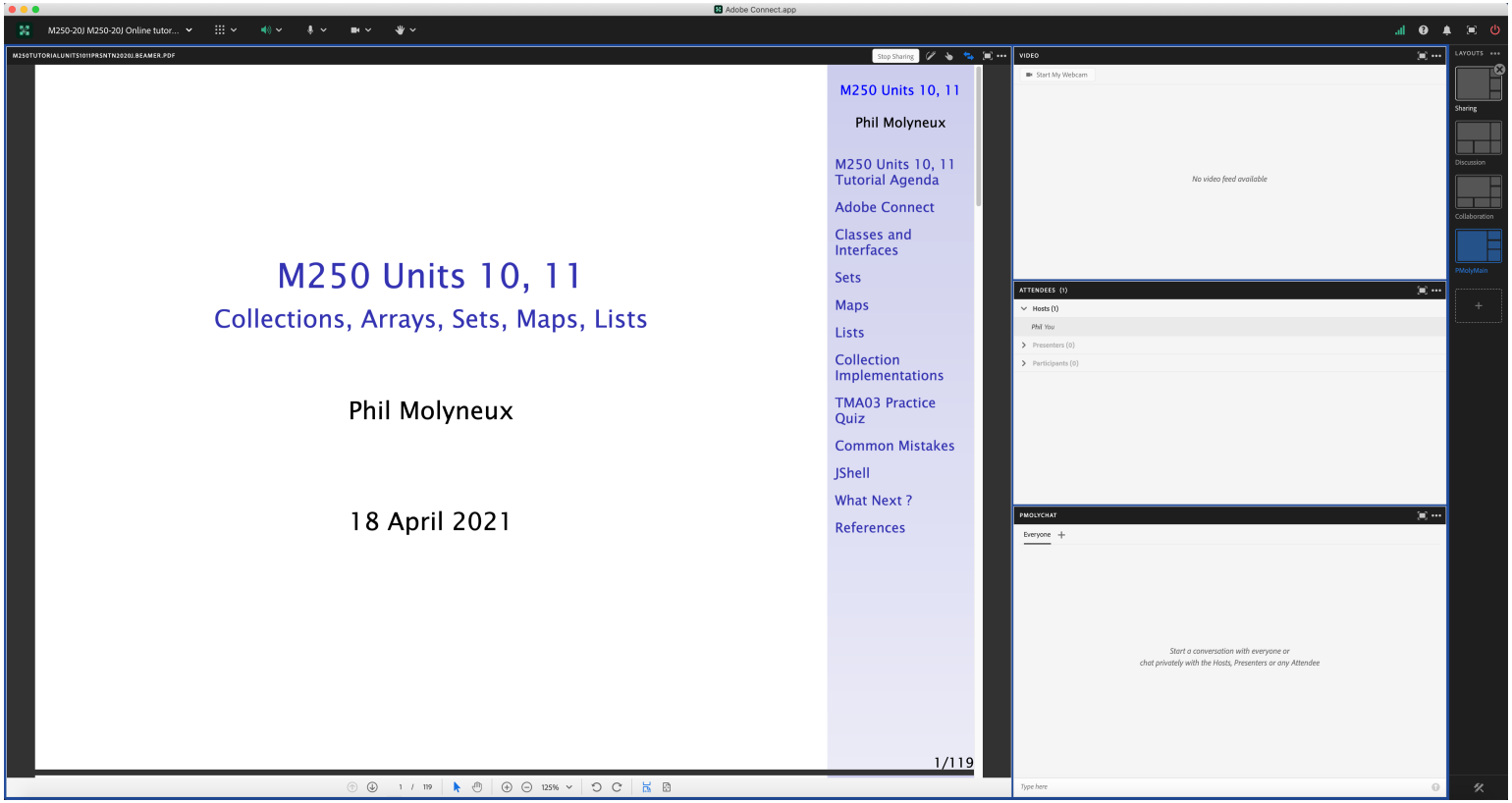
M250 Tutorial

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 ?*

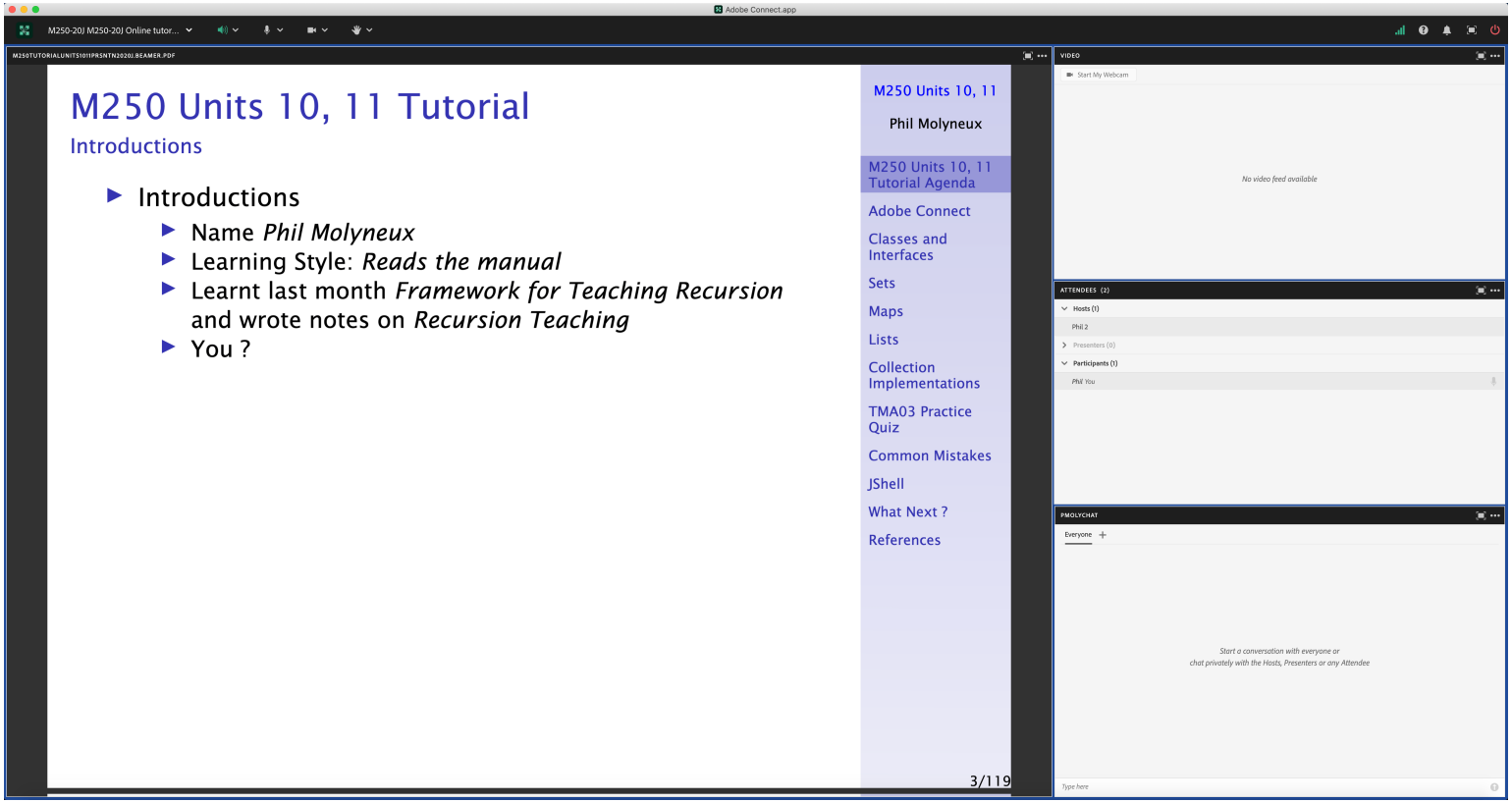
Adobe Connect

Interface — Host View



Adobe Connect

Interface — Participant View



Adobe Connect

Settings

- **Everybody** *Menu bar* › *Meeting* › Speaker & Microphone Setup
- *Menu bar* › *Microphone* › Allow Participants to Use Microphone ✔
- Check Participants see the entire slide **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** ⌘ + \

Adobe Connect Interface

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

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

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

Adobe Connect

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 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 — name it *PMolyChat* — and resize/reposition

Adobe Connect

Layouts

- 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)

Adobe Connect

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

Graphics Conversion

PDF to PNG/JPG

- 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 ⌘ + ⏎

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 and Haskell
- Recursive and iterative versions
- Mergesort, Quicksort and Bubble sort in the same framework
- Sorting via a data structure — *Tree sort*
- Review of Web sites and sorting algorithms used in practice

Sorting Algorithms

Motivation for Studying

- From Knuth (1998, page v)
- . . . 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 ?

Sorting Algorithms

Demonstration 1 Sorting Algorithms as Dances

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

Sorting Algorithms

Activity 1 Card Sorting Exercise (1)

- 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:

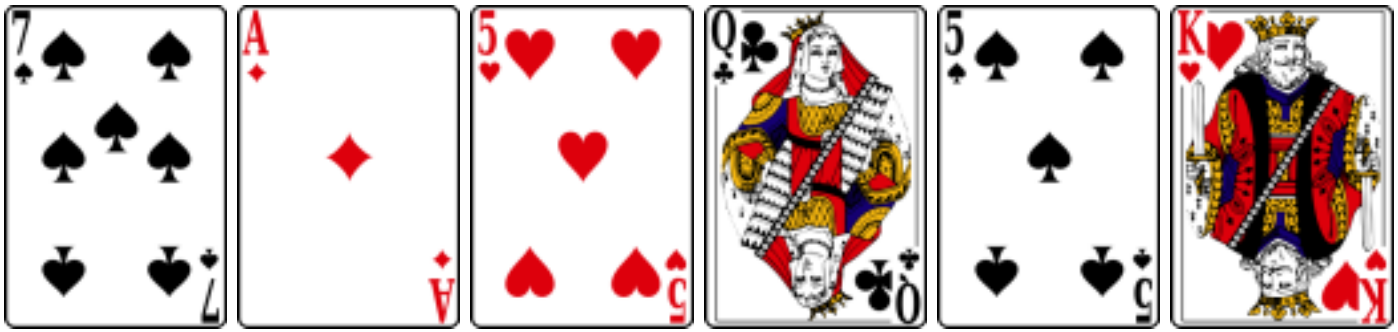
```
Clubs < Diamonds < Hearts < Spades

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.

Sorting Algorithms

Activity 1 Card Sorting Exercise (2)

- 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

Activity 1 Card Sorting Exercise

Working Space

Activity 1 Card Sorting Exercise (3)

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/

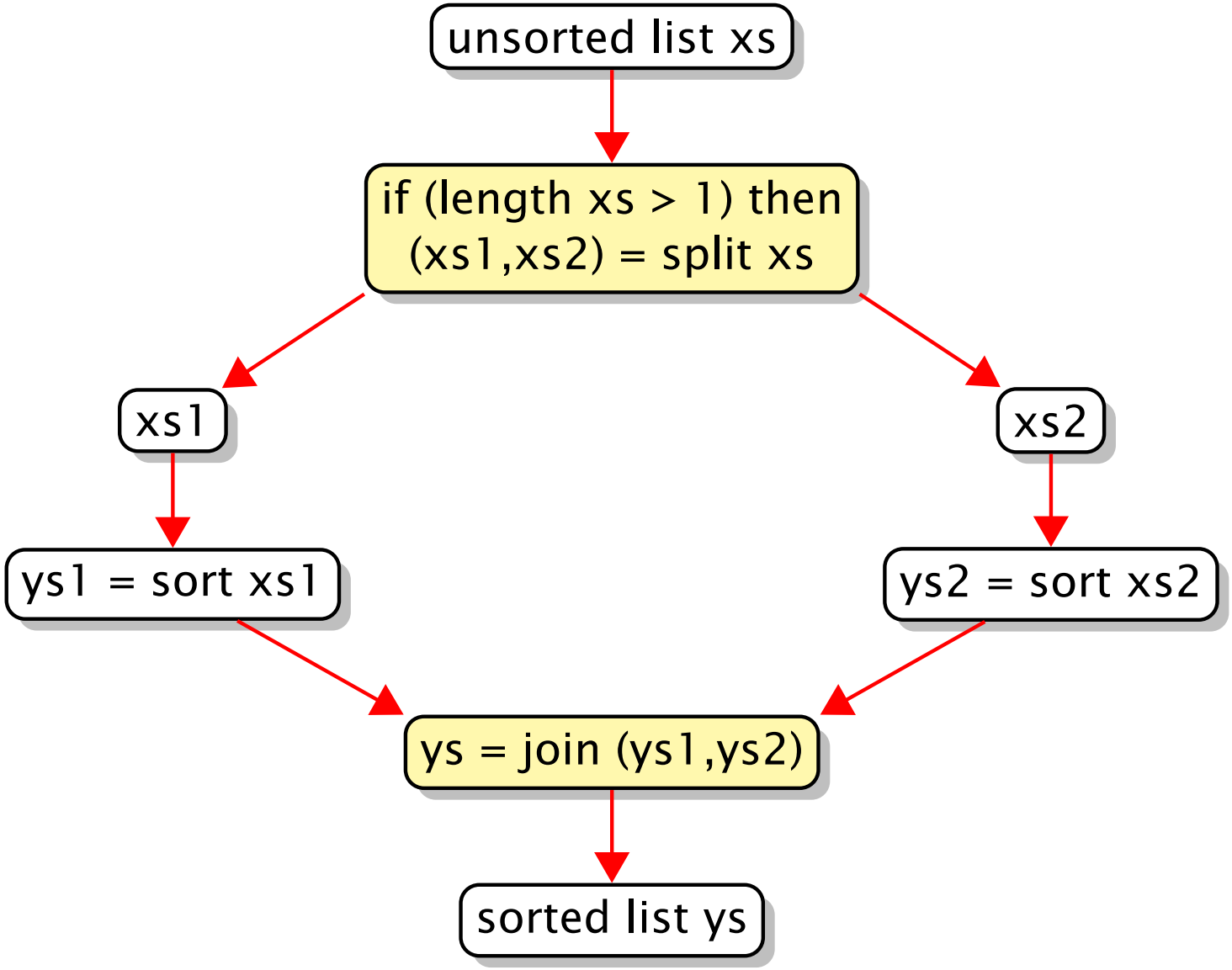
Taxonomy of Sorting Algorithms

Comparison Sorts

- 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, 1997) and Azmoodeh (1990, chp 9), we classify the divide and conquer sorting algorithms by easy/hard split/join
- see diagram below

Taxonomy of Sorting Algorithms

Abstract Sorting Algorithm



Sorting Algorithms

Other Classifications

- 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*
  - Timsort by Tim Peters — used in Python and Java — combination of merge and insertion sorts
  - Haskell — modified Mergesort by Ian Lynagh in GHC implementation

Recursion

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.

Recursion

Evaluation of `fac(6)`

	Expression to Evaluate	Reason
	`fac(6)`	Initial
→	`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

Recursion

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()` use `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

Recursion

Evaluation of `facIter(6)`

	Expression to Evaluate	Reason
	`facIter(6)`	Initial
→	`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

Recursion and Iteration

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

Recursion and Iteration

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
```

Recursion

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 the 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.

Recursion

Structured Programming, GOTO and Recursion

- Bohm & 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, 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

Recursion

Structured Programming, GOTO and Recursion (2)

- 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
- Alan Perlis (1982) *Epigrams on Programming*: [Functional programmers] know the value of everything but the cost on nothing
- Erik Meijer (1991) *Recursion is the GOTO of functional programming*
- Leading to common patterns of higher order functions, `map`, `filter`, `fold` and polymorphic data types

Split/Join Sorting Algorithms

Example Algorithms & Implementation

- Insertion Sort
- Selection Sort
- Merge Sort
- Quicksort
- Bubble Sort
- Implementations in Python, recursive and non-recursive
- Implementations in Haskell — for comparison, optional
- Sorting via data structure — Treesort, Heap Sort

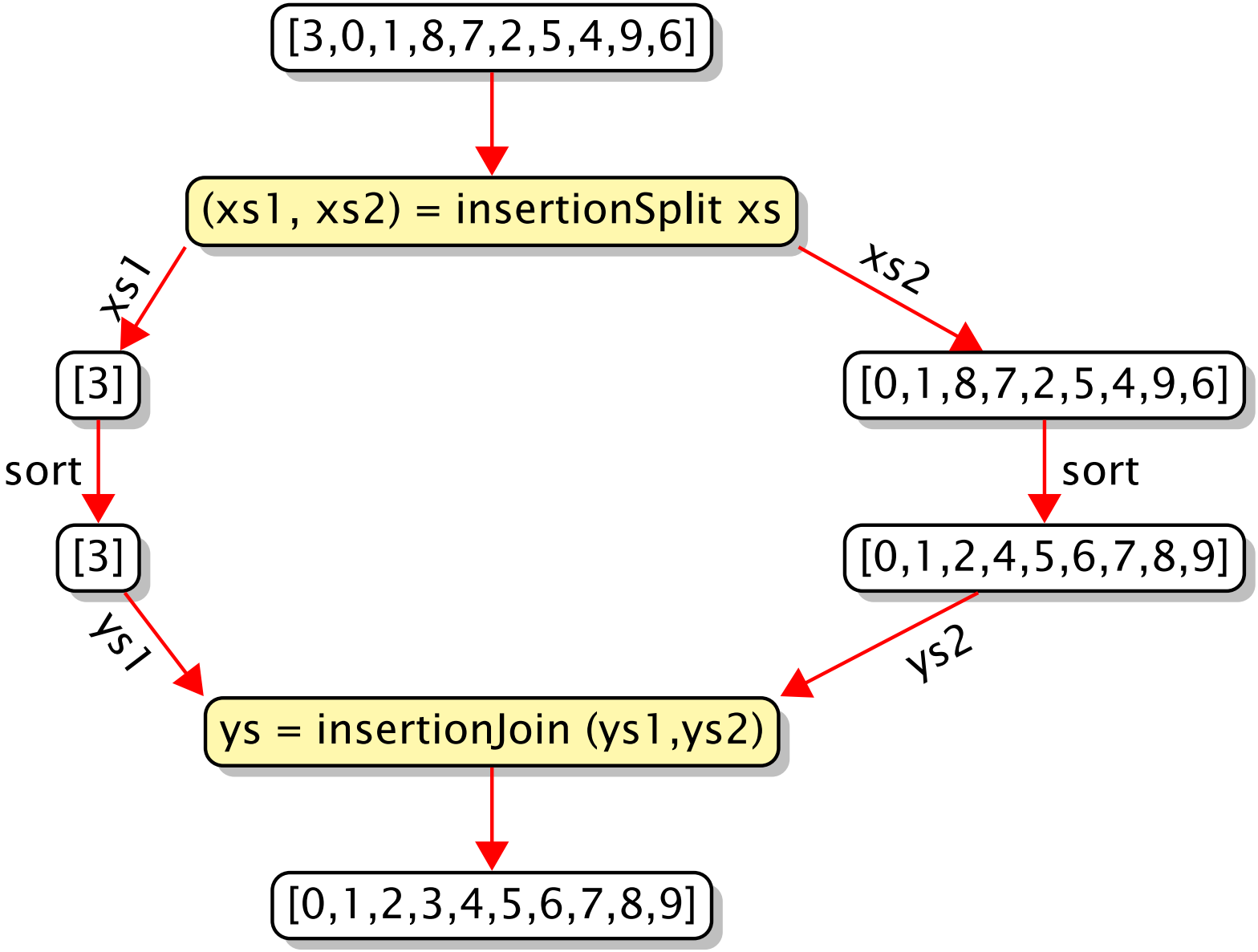
Insertion Sort

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

Insertion Sort

Abstract Sorting Algorithm Diagram



Insertion Sort

Python Implementation

```
4 def insSort(xs) :
5   if len(xs) <= 1 :
6     return xs
7   else :
8     return ins(xs[0],insSort(xs[1:]))

10 def ins(x,xs) :
11   if xs == [] :
12     return [x]
13   elif x <= xs[0] :
14     return [x] + xs
15   else :
16     return [xs[0]] + ins(x,xs[1:])
```

Insertion Sort

Python Rewritten

- In the style of the abstract algorithm

```
20 def insSort01(xs) :
21   if len(xs) <= 1 :
22     return xs
23   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:])
```

Insertion Sort

Haskell Implementation (1)

```haskell
module M269TutorialSorting where
  import Data.List
  import Data.Maybe
```

1. A Haskell script starts with a module header which starts with the reserved identifier, `module` followed by the module name, `M269TutorialSorting`
2. The module name must start with an upper case letter and is the same as the file name (without its extension of *.lhs*)
3. Haskell uses *layout* (or the *off-side rule*) to determine scope of definitions, similar to Python
4. The body of the module follows the reserved identifier `where` and starts with two `import` declarations
5. These import the built-in libraries `Data.List` and `Data.Maybe`

Insertion Sort

Haskell Implementation (2)

```
5  insSort  []  = []
6  insSort  [x] = [x]
7  insSort (x : xs)  =  ins x (insSort xs)

9  ins  x  []  =  [x]
10 ins  x  (y:ys)
11   =  if x <= y
12      then x:y:ys
13      else y : (ins x ys)
```

- For *structured English*, I have used a subset of *Haskell* (http://haskell.org) — in the code above:
- *insSort* and *ins* are function defined by several equations
- We use *indentation* to determine scope — see Landin (1966) and Python (see Python Tutorial: Introduction: First Steps Towards Programming)

Insertion Sort

Haskell Implementation (3)

- Function application is denoted by juxtaposition and is more tightly binding than (almost) anything else
  - we write `f x` and not `f (x)`
  - `f x y` means `(f x) y`

  This notational convention has huge advantages — discuss and also see http://en.wikipedia.org/wiki/Curried_function and http://slid.es/gsklee/functional-programming-in-5-minutes (which does it in JavaScript, worth a look)
- Lists are denoted with brackets `[1,2,3]`, the empty list is `[]`
- `(:)` is the operator that prefixes an element to a list, `1:[2,3] == [1,2,3]`
- Parentheses over-ride precedence

Activity 2

Trace an Evaluation — Haskell

- Evaluation of `insSort [3,0,1,8,7]`
- `Answer goes here`

Activity 2

Trace an Evaluation — Haskell

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

	Expression to Evaluate	Reason
	`insSort [3,0,1,8,7]`	Initial
→	`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 6
→	`ins 3 (ins 0 (ins 1 (7:(ins 8 []))))`	line 13
→	`ins 3 (ins 0 (ins 1 (7:[8])))`	line 9
→	`ins 3 (ins 0 (ins 1 [7,8]))`	(:) operator
→	`ins 3 (ins 0 (1:7:[8]))`	line 12
→	`ins 3 (ins 0 [1,7,8])`	(:) operator
→	`ins 3 (0:1:[7,8])`	line 12
→	`ins 3 [0,1,7,8]`	(:) operator
→	`0:(ins 3 [1,7,8])`	line 13
→	`0:(1:(ins 3 [7,8]))`	line 13
→	`0:(1:(3:7:[8]))`	line 12
→	`[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.

Activity 2 Trace an Evaluation

Insertion Sort — Python Recursive

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

Activity 2 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
→	`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.

Insertion Sort

Non-recursive Implementation

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

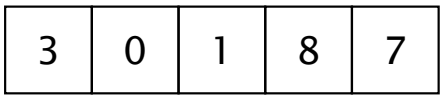
```
for index = 1 to (len(xs)-1) do
  insert xs[index] in order in xs[0..index-1]
```

- Here is a Python implementation of the above (based on Miller and Ranum (2011, page 215)).

```
42 def insertionSort(xs) :
43   for index in range(1, len(xs)) :
44     currentValue = xs[index]
45     position = index
46     while (position > 0) and xs[position - 1] > currentValue :
47       xs[position] = xs[position - 1]
48       position = position - 1

50     xs[position] = currentValue
```

Activity 3

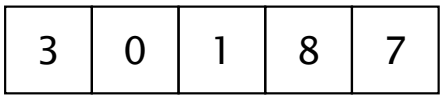
Trace an Evaluation — Python Non-recursive

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

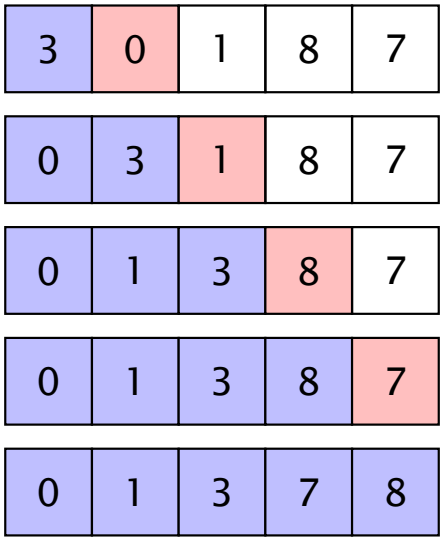
start array
- `Answer goes here`

Activity 3

Trace an Evaluation — Python Non-recursive

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

start array



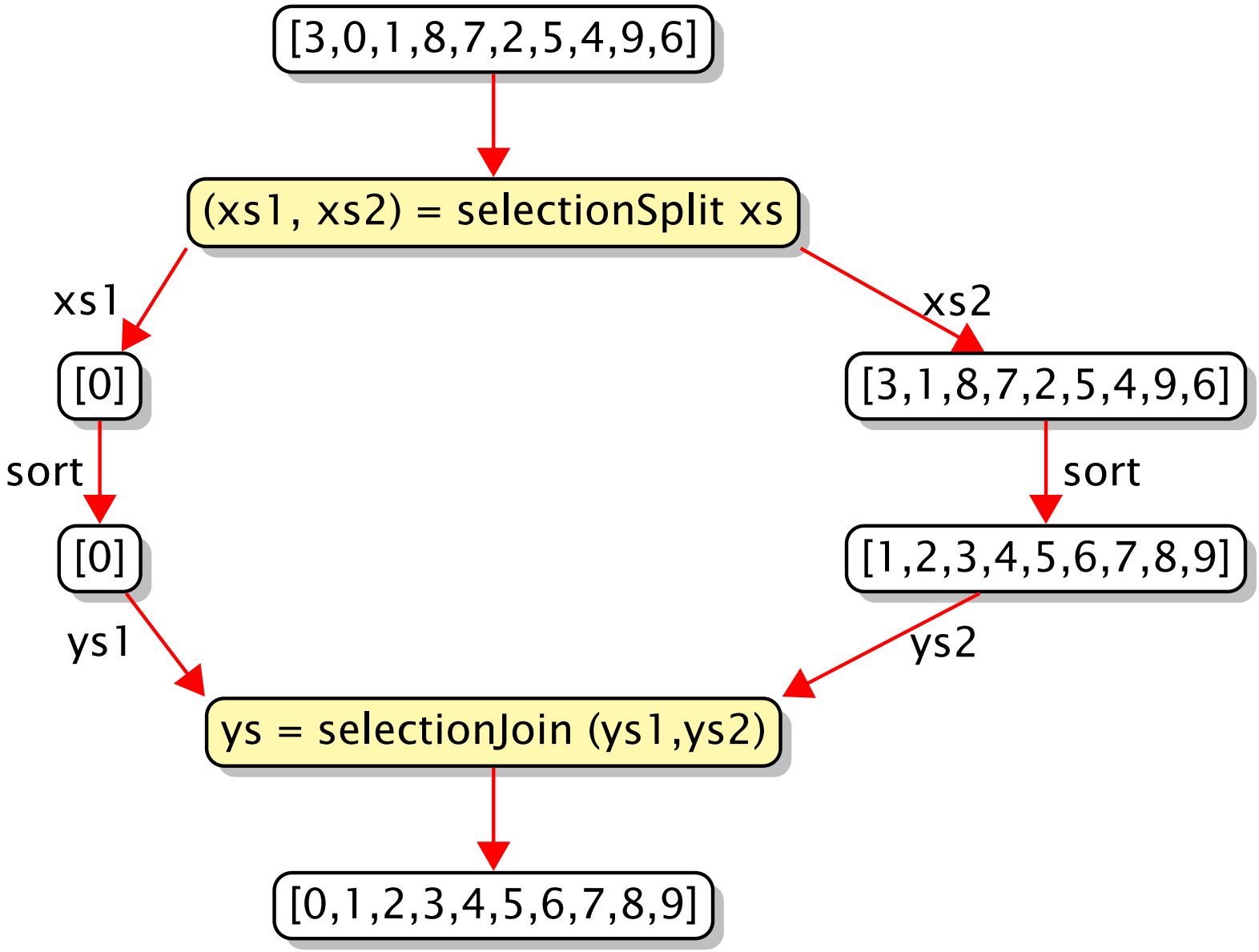
Selection Sort

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

Selection Sort

Abstract Sorting Algorithm Diagram



Selection Sort

Haskell Implementation

```
15  selSort  []  =  []
16  selSort  [x] =  [x]
17  selSort  xs  =  minItem : selSort (xs \\ [minItem])
18    where
19    minItem  =  minimum xs
```

- Explanation of the above:
- `(\\)` is the *list difference* operator
- `[2,1,3,1] \\ [1] == [2,3,1]`
- `minimum` is the Haskell built in function that takes a list a returns the smallest item.
- See the `Data.List` library
- Exercise: produce your own implementation of `minimum` — remember to give it a different name

Activity 4 — Selection Sort

Trace an Evaluation — Haskell

- Evaluation of `selSort [3,0,1,8,7]`
- `Answer goes here`

Activity 4 — Selection Sort

Trace an Evaluation — Haskell

- Evaluation of `selSort [3,0,1,8,7]`

	Expression to Evaluate	Reason
	`selSort [3,0,1,8,7]`	Initial
→	`0 : (selSort [3,1,8,7])`	line 17
→	`0 : (1 : (selSort [3,8,7]))`	line 17
→	`0 : (1 : (3 : (selSort [8,7])))`	line 17
→	`0 : (1 : (3 : (7 : (selSort [8]))))`	line 17
→	`0:(1:(3:(7:[8])))`	line 16
→	`[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.

Selection Sort

Python Implementation

```
54 def selSort(xs) :
55   if len(xs) <= 1 :
56     return xs
57   else :
58     minElmnt = min(xs)
59     minIndex = xs.index(minElmnt)
60     xsWithoutMin = xs[:minIndex] + xs[minIndex+1:]
61     return [minElmnt] + selSort(xsWithoutMin)
```

- Why do we not use `xs.remove(min(xs))` ?

Selection Sort

Non-recursive Implementation

- 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
69
70     # if fillSlot != minIndex: # only swap if different
71     xs[fillSlot],xs[minIndex] = xs[minIndex],xs[fillSlot]
```

Selection Sort

Non-recursive Implementation

- 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.

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.

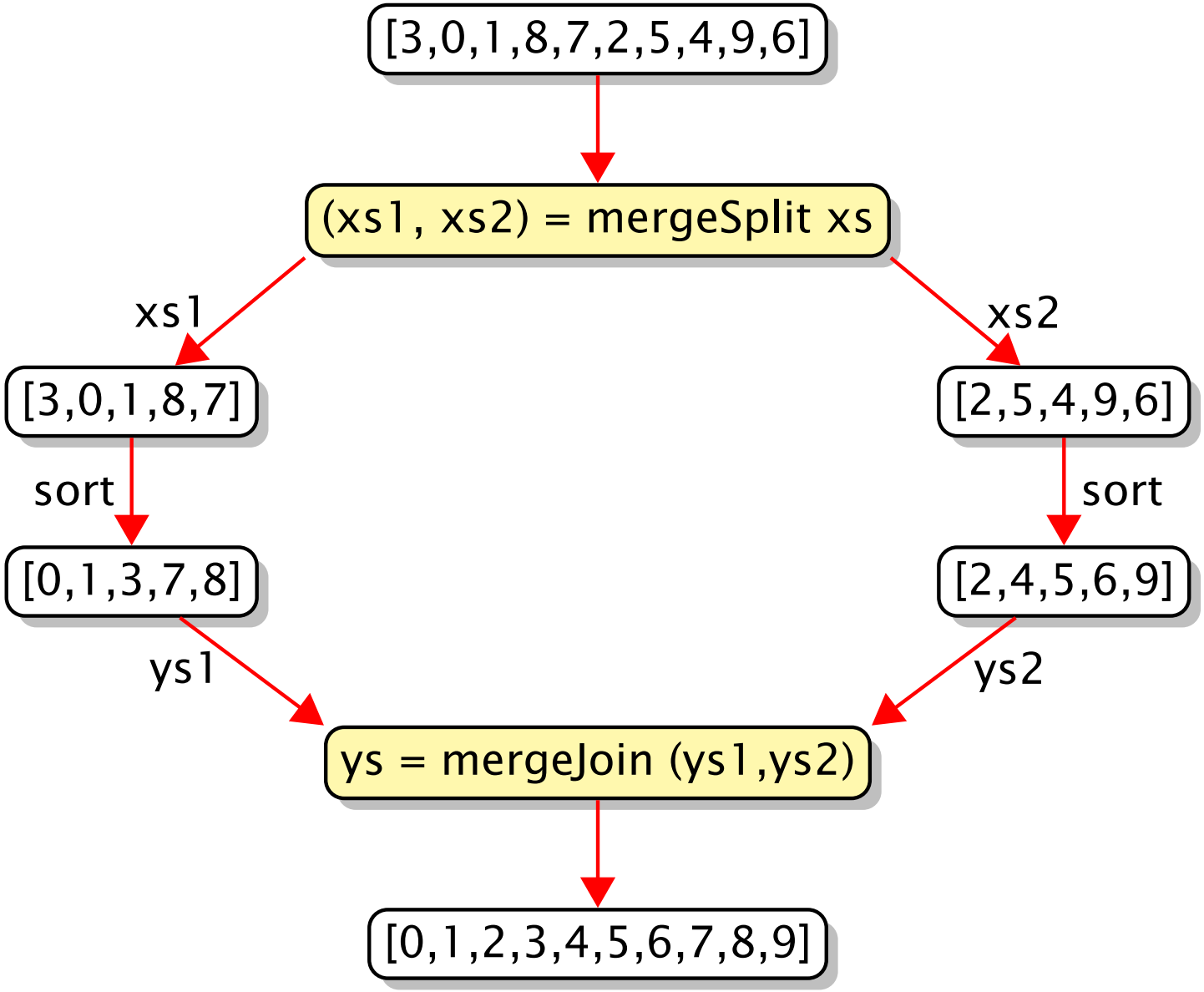
Merge Sort

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

Merge Sort

Abstract Sorting Algorithm Diagram



Merge Sort

Haskell Implementation

```
21  mergeSort  []  =  []
22  mergeSort  [x]  =  [x]
23  mergeSort  xs
24    =  mergeJoin  (mergeSort as) (mergeSort bs)
25       where
26       (as,bs) = mergeSplit xs

28  mergeSplit = mergeSplit2

30  mergeSplit2  xs   =  (take half xs, drop half xs)
31    where
32    half = (length xs) `div` 2

34  mergeJoin  []  ys  =  ys
35  mergeJoin  xs  []  =  xs
36  mergeJoin  (x:xs)  (y:ys)
37    |  x <= y     =  x : mergeJoin xs (y:ys)
38    |  otherwise  =  y : mergeJoin (x:xs) ys
```

Haskell Implementation

Code Description

- Reserved words and built in function are in `blue`
- `take n xs` returns the first `n` of `xs` as a new list
- `div` is the integer division function, the back quotes make it an infix operator
- `3 ‘div‘ 2 == div 3 2 == 1`
- In `mergeJoin`, if the boolean expression following a vertical bar (`|`) evaluates to `True` then the value of the left hand side is given by the expression on the right of the following “=” — the lines are known as *guards* and are evaluated in turn until one is found to be true (`otherwise` is a nickname for `True`)
- We have `mergeSplit1` and `mergeSplit2` to illustrate choices.
- The code for `mergeSplit1` is given below — it splits the list with just one traversal of the list

Merge Sort

Haskell `mergeSplit1`

```
40  mergeSplit1  []   =  ([],[])
41  mergeSplit1  [x]  =  ([x],[])
42  mergeSplit1  (x:y:zs)
43    =  (x:xs, y:ys)
44       where
45       (xs,ys) = mergeSplit1 zs
```

- `mergeSplit1` recursively splits the list by adding alternate elements to the two parts of the result pair
- The code in Python would look similar

Merge Sort

Python Implementation

```
86 def mergeSort(xs) :
87   if len(xs) <= 1 :
88     return xs
89   else :
90     (aList,bList) = mergeSplit(xs)
91     return mergeJoin(mergeSort(aList),mergeSort(bList))

93 def mergeSplit(xs) :
94   return mergeSplit2(xs)

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

100 def mergeJoin(xs,ys) :
101   if xs == [] :
102     return ys
103   elif ys == [] :
104     return xs
105   elif xs[0] <= ys[0] :
106     return [xs[0]] + mergeJoin(xs[1:],ys)
107   else :
108     return [ys[0]] + mergeJoin(xs,ys[1:])
```

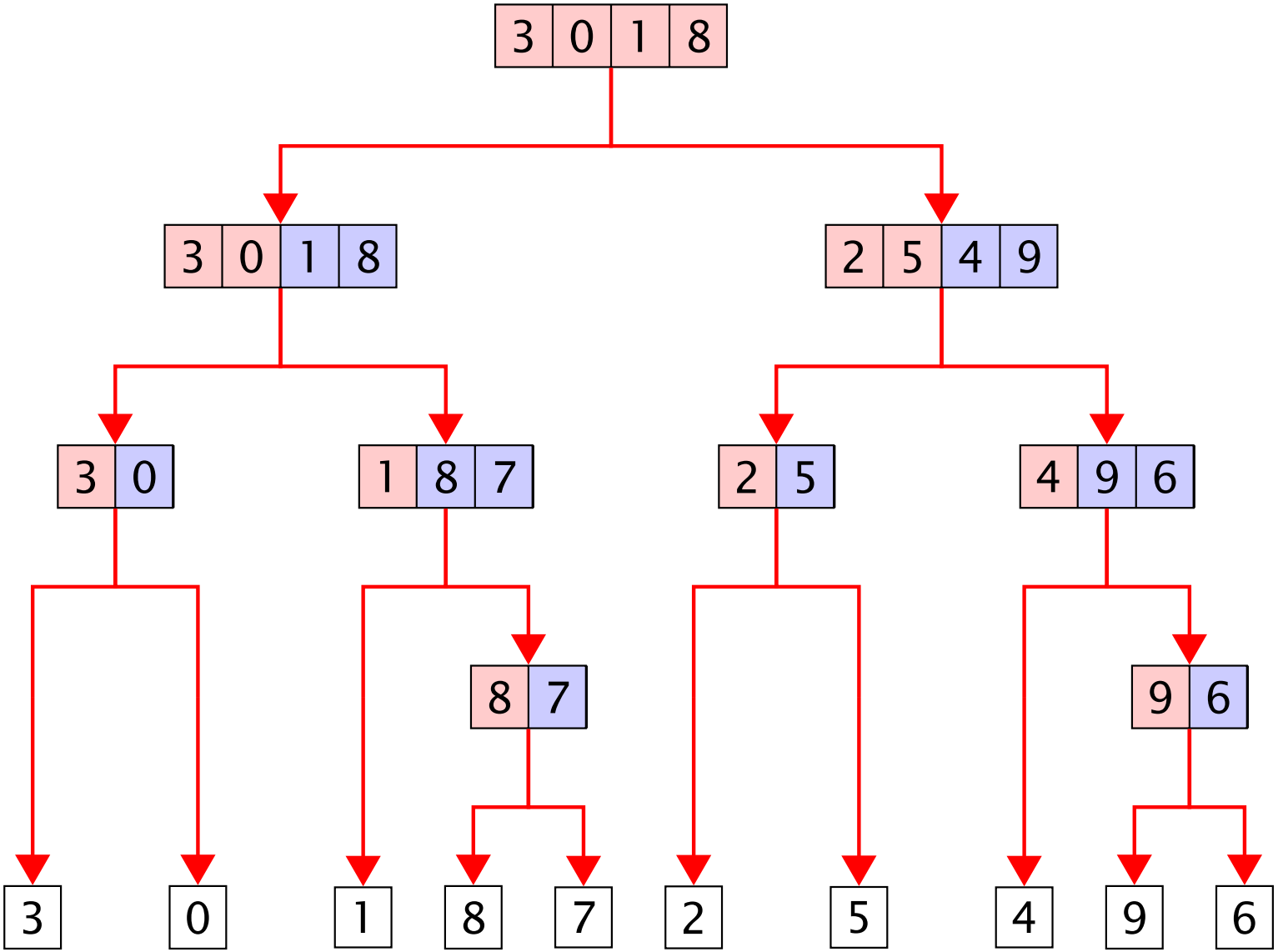
Merge Sort

Python mergeSplit1

```
110 def mergeSplit1(xs) :
111   if len(xs) == 0 :
112     return  ([],[])
113   elif len(xs) == 1 :
114     return (xs,[])
115   else :
116     (aList,bList) = mergeSplit1(xs[2:])
117     return ([xs[0]] + aList, [xs[1]] + bList)
```

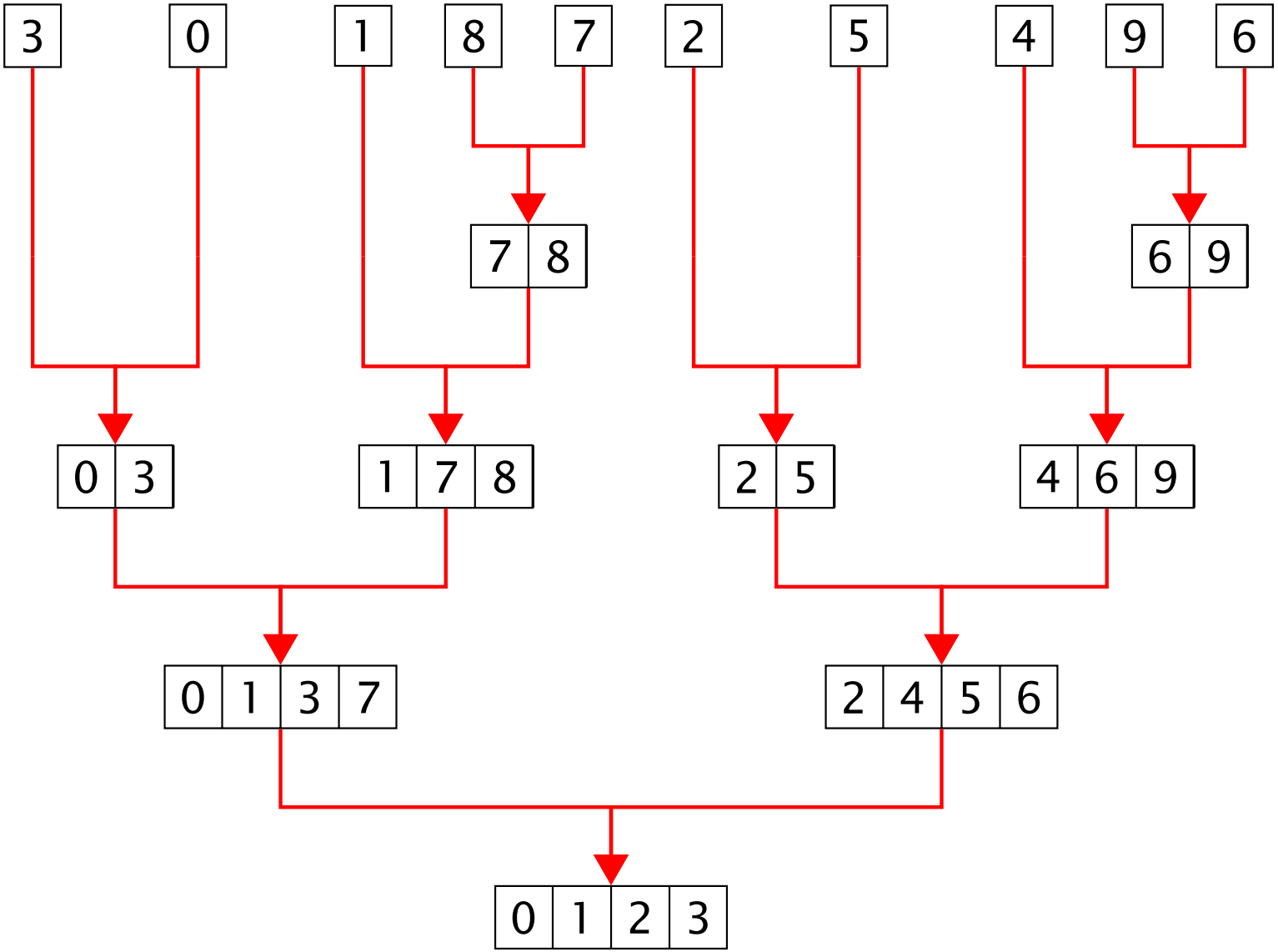
Merge Sort Diagram

Merge Sort Split Phase



Merge Sort Diagram

Merge Sort Join Phase



Merge Sort

Python In-Place (1)

- 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

```
119 def mergeSortInPlace(xs) :
120   if len(xs) > 1 :
121     print("Splitting ", xs)
122   else :
123     print("Singleton ", xs)

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

129     mergeSortInPlace(aList)
130     mergeSortInPlace(bList)
```

Merge Sort

Python In-Place (2)

```
132     i,j,k = 0,0,0
133     while i < len(aList) and j < len(bList) :
134       if aList[i] < bList[j] :
135         xs[k] = aList[i]
136         i = i + 1
137       else :
138         xs[k] = bList[j]
139         j = j + 1
140       k = k + 1

142     while i < len(aList) :
143       xs[k] = aList[i]
144       i = i + 1
145       k = k + 1

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

Merge Sort

Python In-Place (3)

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

```
152   if len(xs) > 1 :
153     print("Merging ", aList, ",", bList, "to", xs)
154   else :
155     print("Merged ", xs)
```

Merge Sort

Python Code Description

- `␣` 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.

Merge Sort

Python In-Place (3)

- 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]
```

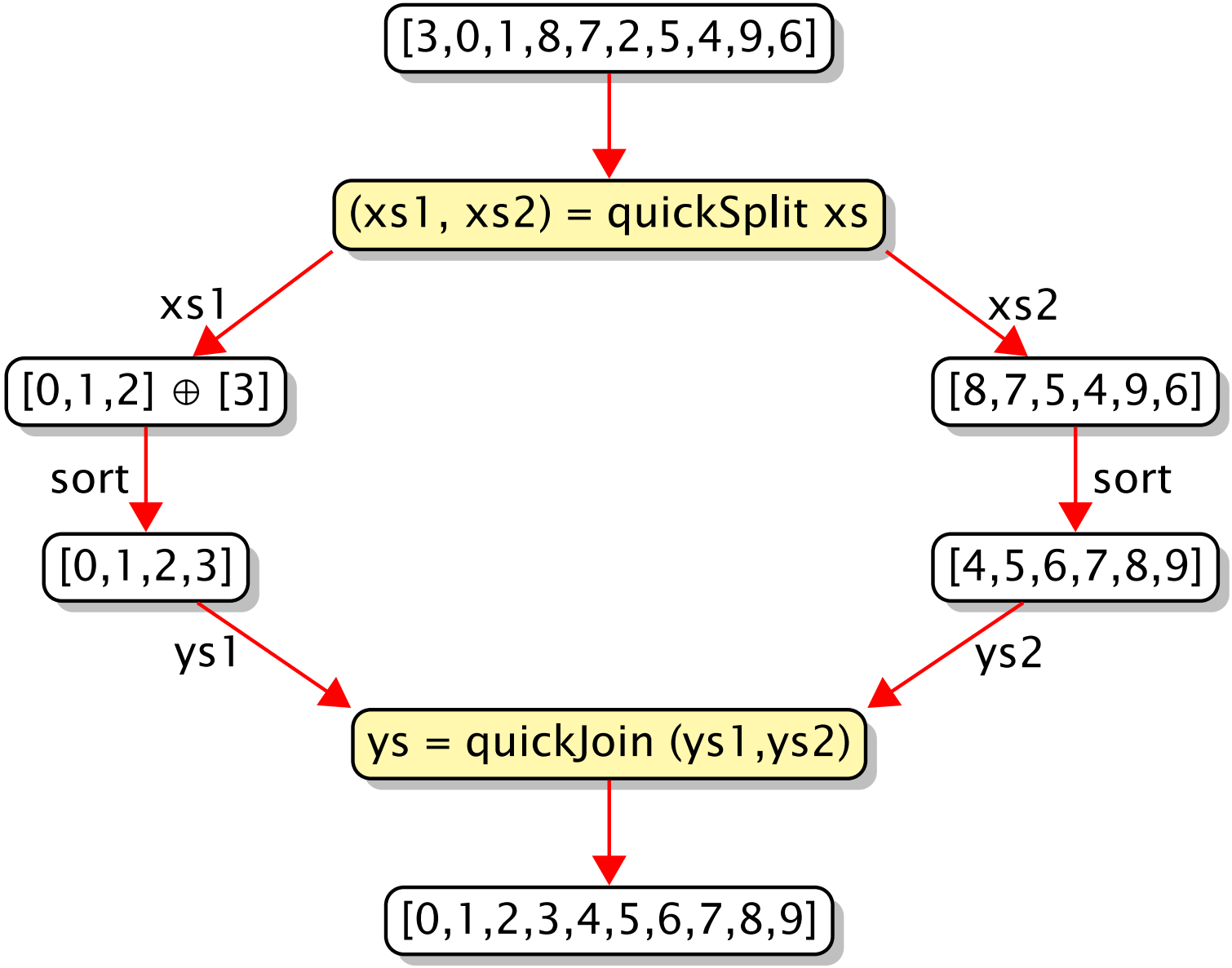
Quicksort

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

Quicksort

Abstract Sorting Algorithm Diagram



Quicksort

Haskell Implementation

```
47  quickSort1  []  =  []
48  quickSort1  (x:xs)
49    =  quickSort1 [y | y <- xs, y < x]
50       ++ [x]
51       ++ quickSort1 [y | y <- xs, y >= x]
```

- This uses the Haskell version of the list comprehension notation
- Based on classical set notation and originally implemented in *Miranda* out of David Turner in 1983–6 (see http://miranda.org.uk)
- This idea is available in Python but in a slightly different syntax
- ++ is the list append operator — denoted $\oplus$ in various courses and texts

List Comprehensions

In Haskell and Python

- 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

Quicksort

Python

```
159 def qsort(xs) :
160   if not xs :
161     return []
162   else :
163     pivot = xs[0]
164     less = [x for x in xs     if x <  pivot]
165     more = [x for x in xs[1:] if x >= pivot]
166     return qsort(less) + [pivot] + qsort(more)
```

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

Quicksort

Python In-Place (1)

- 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

Quicksort

Python In-Place (2)

- 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)
```

Quicksort

Python In-Place (3)

```
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
```

Quicksort

Python In-Place (4)

- 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.

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.

Bubble Sort

Haskell

- Here is a naive version (based on http://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort#Haskell
- it is *naive* because it does the check for changes in a simple way.
- See the above Web site for more sophisticated versions

```
53  bubbleSort  xs
54    =  if (ts == xs) then ts else (bubble ts)
55       where
56       ts = bubble xs

58  bubble []  = []
59  bubble [x] = [x]
60  bubble (x1:x2:xs)
61    |  x1 > x2    =  x2 : (bubble (x1 : xs))
62    |  otherwise  =  x1 : (bubble (x2 : xs))
```

Bubble Sort

Haskell Code Description

- The expression `(x1:x2:xs)` denotes a list of at least two items whose first two items are `x1` and `x2` and the rest of the list is `xs`
- The third equation defining `bubble` uses *boolean guards* starting with (`|`) rather than a *conditional expression* (`if ... then ... else ...`)
- it could be written the other way and remove the need to understand this style of function declaration but this is a frequently used style in Haskell

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)

```
203 def bubbleSort(xs) :
204   for passNum in range(len(xs) - 1, 0, -1) :
205     for i in range(passNum) :
206       if xs[i] > xs[i+1] :
207         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]`)

What Next ?

Trees, Graph algorithms, Greed, Logic, Computability

- Binary trees, Binary heaps and Heap sort
- Searching — searching for patterns, string searches
- Hashing and hash tables
- Binary search trees, height balanced binary search trees, AVL trees
- Graph algorithms
- Greedy algorithms
- Sunday 6 February 2022 Tutorial Online Binary Trees, Graph algorithms
- Logic, Computability
- Sunday 13 March 2022 Tutorial Online Logic
- Sunday 24 April 2022 Tutorial Online Computability

Tree Sort

Abstract Algorithm

- *Build Binary Search Tree* — build a binary search tree from the list of keys to be sorted
- *Traverse Tree In-Order* — traverse the tree in-order to output the keys in sorted order

Tree Sort

Python (1)

```
211 from collections import namedtuple

213 EmptyTreeBT = None

215 NodeBT = namedtuple('NodeBT'
216                       ,['dataBT','leftBT','rightBT'])

218 # Binary Tree Operations

220 def makeEmptyBT() :
221   return EmptyTreeBT

223 def makeBT(x,t1,t2) :
224   return NodeBT(x,t1,t2)

226 def isEmptyBT(t) :
227   return t is EmptyTreeBT
```

Tree Sort

Python (2)

- This is from `SortingPython.py`
- Reserved identifiers are shown in `this color`
- User defined data constructors such as `NodeBT` and `EmptyTreeBT` are shown in `that color`
- `NodeBT` is a named tuple with named fields — a *quick and dirty* object
- `makeEmptyBT`, `makeBT` are *constructor* functions — we could have used the raw named tuple and `None` but the discipline is good for you
- `isEmptyBT` uses the `is` operator for identity check (not (==))
- **Health Warning:** these notes may not be totally consistent with syntax colouring.

Tree Sort

Python (3)

- `insertListBST` and `insertBST` insert a list of items into a Binary Search Tree
- To be consistent, we should have used the constructor functions to hide the implementation.

```
274 def insertBST(x,t) :
275   if isEmptyBT(t) :
276     return NodeBT(x,EmptyTreeBT,EmptyTreeBT)
277   else :
278     y = t.dataBT
279     if x < y :
280       return NodeBT(y, insertBST(x,t.leftBT),t.rightBT)
281     elif x > y :
282       return NodeBT(y, t.leftBT, insertBST(x,t.rightBT))
283     else :
284       return t

286 def insertListBST(t,xs) :
287   if xs == [] :
288     return t
289   else :
290     return insertListBST(insertBST(xs[0],t),xs[1:])
```

Tree Sort

Python (4)

- `inOrderBT` takes a Binary Tree and does an in-order traversal
- `treeSort` combines `insertListBST` and `inOrderBT`

```
251 def inOrderBT(t) :
252   if isEmptyBT(t) :
253     return []
254   else :
255     return (inOrderBT(t.leftBT) + [t.dataBT]
256             + inOrderBT(t.rightBT))
```

```
292 def treeSort(xs) :
293   return inOrderBT(insertListBST(makeEmptyBT(),xs))
```

Tree Sort

Python (5)

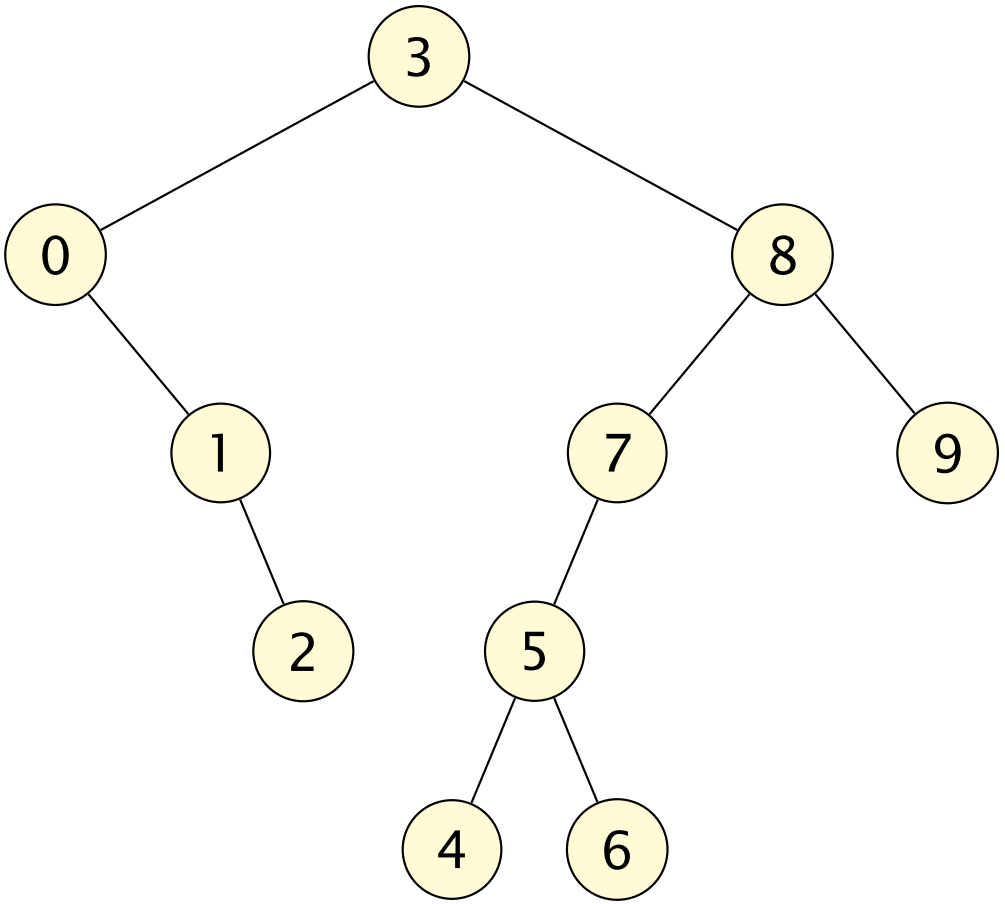
- Example list and tree

```
297 xs = [3,0,1,8,7,2,5,4,9,6]

299 egTree = insertListBST(makeEmptyBT(),xs)

301 egTreeTest = NodeBT(3,
302                NodeBT(0,
303                  EmptyTreeBT,
304                  NodeBT(1,
305                    EmptyTreeBT,
306                    NodeBT(2, EmptyTreeBT, EmptyTreeBT))),
307                NodeBT(8,
308                  NodeBT(7,
309                    NodeBT(5,
310                      NodeBT(4, EmptyTreeBT, EmptyTreeBT),
311                      NodeBT(6, EmptyTreeBT, EmptyTreeBT)),
312                    EmptyTreeBT),
313                  NodeBT(9, EmptyTreeBT, EmptyTreeBT)))
```

Example Tree Sort 1

Insert `[3,0,1,8,7,2,5,4,9,6]` into `EmptyTreeBT`

`egTreeTest`



- The in-order traversal of `egTreeTest` outputs
- `[0,1,2,3,4,5,6,7,8,9]`

Tree Sort

Haskell (1)

```
64  data BinTree a = EmptyTreeBT
65                 | NodeBT a (BinTree a) (BinTree a)
66                 deriving (Eq,Ord,Show,Read)

68  -- BSTree is an alias for BinTree,
69  -- we have to enforce the Binary Search Tree property

71  type BSTree a = BinTree a
```

- The code starting with data (line 64) is an *Algebraic Datatype* declaration. Algebraic datatypes allow you just to name things and use them in your program
- Meta-magic and avoids ever needing to use pointers
- For a description see Algebraic data type and Marlow and Peyton Jones (2010, section 4.2.1)
- -- comments out a line

Tree Sort

Haskell (2)

- `BinTree` is the name of the type and `EmptyTreeBT`, `NodeBT` are the two *data constructors*
- `a` is a *type variable* — that is, a variable that ranges over types (not values). It could be any type (subject to any restrictions we place on it): primitive types such as `Int`, `Bool`, built-in structured types such as tuples or list, or other user defined types
- The constructor `EmptyTreeBT` is to represent an empty tree (took ages to think of that name)
- The constructor `Node` takes three arguments: the first is of type `a` and is meant to represent the data stored at a node, the second and third are of type `BinTree a` and indicate the left and right sub trees

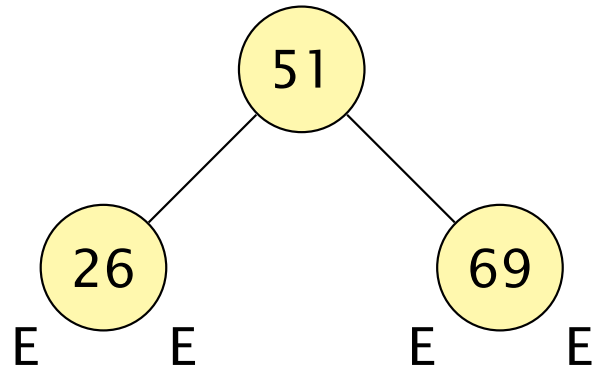
Tree Sort

Haskell (3)

- Here is a sample tree with 51 at the root and left and right subtree with 26 and 69 at their roots

```
NodeBT 51
  (NodeBT 26 EmptyTreeBT EmptyTreeBT)
  (NodeBT 69 EmptyTreeBT EmptyTreeBT)
```

Here is the usual diagram of this tree (with the empty trees labelled as E):



Tree Sort

Haskell (4)

- The `deriving (Eq,Ord,Show,Read)` part of the declaration produces *derived instances* for `BinTree` is the type classes for equality (`Eq`), ordering (`Ord`), printing (`Show`) and reading from files (or standard input) (`Read`)
- Equality as a derived instance is just lexicographic — that is, two trees are equal if and only if they look the same
- `Show` and `Read` do the *fairly* obvious thing — the above example would be printed or read as you see it above.
- The (`|`) is just the syntax separating the two constructors
- The line starting `type` (line `71`) is a *type synonym* declaration — this is not needed apart from making the code a bit more readable (to distinguish Binary Search Trees from other Binary Trees)

Tree Sort

Haskell (5)

```
72  insertBST :: (Ord a) => BSTree a -> a -> BSTree a

74  insertBST EmptyTreeBT x
75    =  NodeBT x EmptyTreeBT EmptyTreeBT

77  -- Note that insertBST does not accept duplicate keys,
78  -- see \citet[page 271]{millar:2011python}

80  insertBST (NodeBT y leftT rightT) x
81    | x < y   =  NodeBT y (insertBST leftT x) rightT
82    | x > y   =  NodeBT y leftT (insertBST rightT x)
83    | x == y  =  NodeBT y leftT rightT

85  insertListBST :: (Ord a) => BSTree a -> [a] -> BSTree a
86  insertListBST t [] = t
87  insertListBST t (x:xs)
88          =  insertListBST (insertBST t x) xs
```

Tree Sort

Haskell (6)

- The line starting `insertBST ::` (line 72) is a *Type Signature* which specifies the type of the function `insertBST`
- `Ord a` is a *context* for the type following `=>` with one *class assertion* — it restricts the type variable `a` to be a member of the `Ord` type class
- `BSTree a -> a -> BSTree a` says that `insertBST` takes a binary tree and an item and returns a binary tree.
- The function type operator `->` is right associative (to match left association of function application) — see Lee (2013).

Tree Sort

Haskell (7)

```
89  inorderBST :: BSTree a -> [a]
90  inorderBST EmptyTreeBT  =  []
91  inorderBST (NodeBT x leftT rightT)
92    =  (inorderBST leftT) ++ [x] ++ (inorderBST rightT)

94  treeSort :: Ord a => [a] -> [a]
95  treeSort xs = inorderBST (insertListBST EmptyTreeBT xs)
```

- The ++ is the list append operator
- treeSort takes a list xs and uses insertListBST to insert the list into EmptyTreeBT and then inorderBST to traverse the tree

Tree Sort

Haskell (8) Alternative Definitions

- Alternative tree building bracketing from the right

```
96  insertBST01 :: (Ord a) => a -> BSTree a -> BSTree a

98  insertBST01 x EmptyTreeBT
99    =  NodeBT x EmptyTreeBT EmptyTreeBT

101 -- Note that insertBST01 does not accept duplicate keys,
102 -- see \citet[page 271]{millar:2011python}

104 insertBST01 x (NodeBT y leftT rightT)
105   | x < y   =  NodeBT y (insertBST01 x leftT) rightT
106   | x > y   =  NodeBT y leftT (insertBST01 x rightT)
107   | x == y  =  NodeBT y leftT rightT

109 insertListBST01 :: (Ord a) => BSTree a -> [a] -> BSTree a
110 insertListBST01 t [] = t
111 insertListBST01 t (x:xs)
112   =  insertBST01 x (insertListBST01 t xs)
```

Tree Sort

Haskell (9) Alternative Definitions

- Some more idiomatic Haskell using *higher order functions*
- (`.`) is the *function composition* operator
- `(f . g) x = f (g x)`
- `foldl` and `foldr` capture common patterns of recursion on lists

```
113  treeSort01 :: Ord a => [a] -> [a]
114  treeSort01 = inorderBST . (insertListBST EmptyTreeBT)
115  -- point free style requires explicit type signature
116  -- because of the monomorphism restriction

118  insertListBSTa :: (Ord a) => [a] -> BSTree a
119  insertListBSTa xs = foldl insertBST EmptyTreeBT xs

121  insertListBST01a :: (Ord a) => [a] -> BSTree a
122  insertListBST01a xs = foldr insertBST01 EmptyTreeBT xs
```

Tree Sort

Haskell (10)

foldl (⊕) z [x_1,x_2,...,x_n]
→ (...((z ⊕ x_1) ⊕ x_2) ⊕...)⊕ x_n
foldr (⊕) z [x_1,x_2,...,x_n]
→ x_1 ⊕ (x_2 ⊕...⊕ (x_n ⊕ z)...)

- Examples
- `sum xs = foldr (+) 0 xs`
- `product xs = foldr (*) 1 xs`
- `concat xss = foldr (++) [] xss`
- Higher order functions tend to get used a lot in idiomatic functional programming
- *Higher order functions* take functions as arguments and/or return functions as results

Tree Sort

Haskell (10)

```
124  xs = [3,0,1,8,7,2,5,4,9,6]

126  egTreeTesta = insertListBSTa xs

128  testA
129    = egTreeTesta
130      == NodeBT 3
131           (NodeBT 0
132              EmptyTreeBT
133              (NodeBT 1
134                 EmptyTreeBT
135                 (NodeBT 2 EmptyTreeBT EmptyTreeBT)))
136           (NodeBT 8
137              (NodeBT 7
138                 (NodeBT 5
139                    (NodeBT 4 EmptyTreeBT EmptyTreeBT)
140                    (NodeBT 6 EmptyTreeBT EmptyTreeBT))
141                 EmptyTreeBT)
142              (NodeBT 9 EmptyTreeBT EmptyTreeBT))
```

Tree Sort

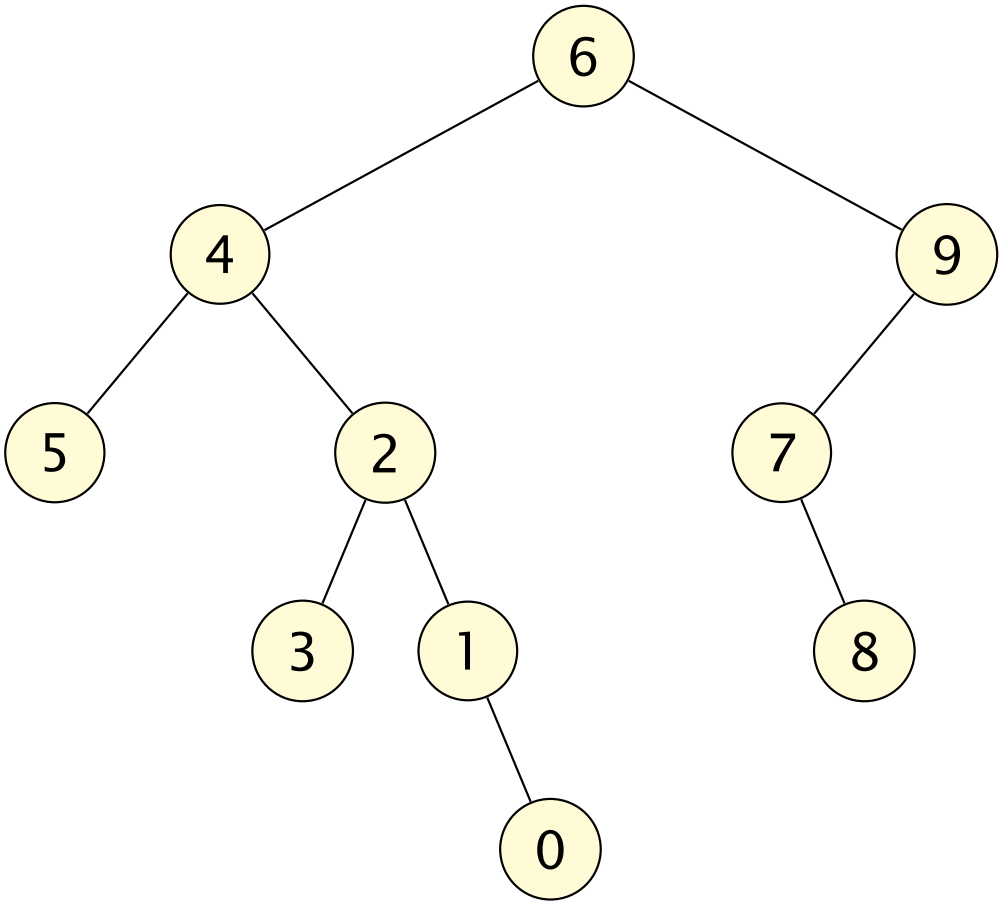
Haskell (11)

```
143  -- xs = [3,0,1,8,7,2,5,4,9,6]

145  egTreeTest01a = insertListBST01a xs

147  test01A
148    = egTreeTest01a
149      == NodeBT 6
150           (NodeBT 4
151              (NodeBT 2
152                 (NodeBT 1
153                    (NodeBT 0 EmptyTreeBT EmptyTreeBT)
154                    EmptyTreeBT)
155                 (NodeBT 3 EmptyTreeBT EmptyTreeBT))
156              (NodeBT 5 EmptyTreeBT EmptyTreeBT))
157           (NodeBT 9
158              (NodeBT 7
159                 EmptyTreeBT
160                 (NodeBT 8 EmptyTreeBT EmptyTreeBT))
161              EmptyTreeBT)
```

Example Tree Sort 2

Insert `[3,0,1,8,7,2,5,4,9,6]` into `EmptyTreeBT`

`egTreeTest01a`



- `egTreeTest01a` is built with `foldr`
- The in-order traversal of `egTreeTest01a` outputs
- `[0,1,2,3,4,5,6,7,8,9]`

Heap Sort

Abstract Algorithm

- A Binary Heap is a Heap using a binary tree with two additional properties:
  - *Compact shape* A binary heap is a complete binary tree — every level, except possibly the last, is completely filled and all nodes in the last level are as for left as possible.
  - *Heap property* All nodes are either greater than or equal to or less than or equal to each of its children.
- In many implementations, the Binary Heap is implemented as an implicit data structure using an array
- The array is a breadth first listing of the nodes
- New nodes can be added in the *next* position in the implicit tree and then percolated or sifted up the tree to its (or a) correct position.
- If the root of the tree is deleted then the *last* node is promoted to the root and percolated or sifted down the tree to a correct place

Heaps

Implementations and Applications

- There are lots of varieties of heaps
- Used later in M269 for Priority queues
- As well as Miller and Ranum and the M269 material, see
  - Comparison of Priority Queue implementations in Haskell
  - Louis Wasserman: Playing with Priority Queues
- TODO: typeset the Python and Haskell for this

Sorting

Web Links

- **Rosetta Code Sorting Algorithms** http://rosettacode.org/wiki/Sorting_algorithms — sorting algorithms implemented n lots of programming languages
- **Sorting Algorithm Animations** http://www.sorting-algorithms.com — 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!

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)

Haskell

Web Links & References

- **Haskell Language** https://www.haskell.org
- **HaskellWiki** https://wiki.haskell.org/Haskell
- **Learn You a Haskell for Great Good!** http://learnyouahaskell.com — very readable introduction to Haskell
- **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
- **Functors, Applicatives, and Monads in Pictures** http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html — a very good outline with cartoons
- **Haskell Wikibook** https://en.wikibooks.org/wiki/Haskell

Sorting Algorithms

Demonstration 2 Sorting Algorithms as Dances

- Quicksort
- https://www.youtube.com/user/AlgoRythmics
- the hats make the point(!)