M269 Sorting

Unit 3

Contents

1 M269 Tutorial Agenda

- Welcome & introductions
- *Tutorial topics:* Unit 3 Sorting Algorithms
- *Key point:* many of the sorting algorithms can be seen as variations on an abstract split/join algorithm
- *Adobe Connect* — if you or I get cut off, wait till we reconnect (or send you an email)
- *Time:* about 2 hours
- Do ask questions or raise points.
- *Source:* of slides, notes, programs and playing cards:
- www.pmolyneux.co.uk/OU/M269FolderSync/M269TutorialNotes/M269TutorialSorting
- *Slides:* M269Prsntn2020JTutorialSorting.beamer.pdf
- *Notes:* M269Prsntn2020JTutorialSorting.article.pdf
- 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

1.1 Motivation

- 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 ?

1.2 Demonstration 1 Sorting Algorithms as Dances

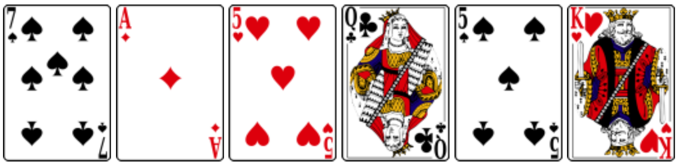
- Insertion Sort
- AlgoRythmics
- This is the folk music that inspired Bartók
- Compare the dance with the Python algorithm for Insertion Sort below

1.3 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

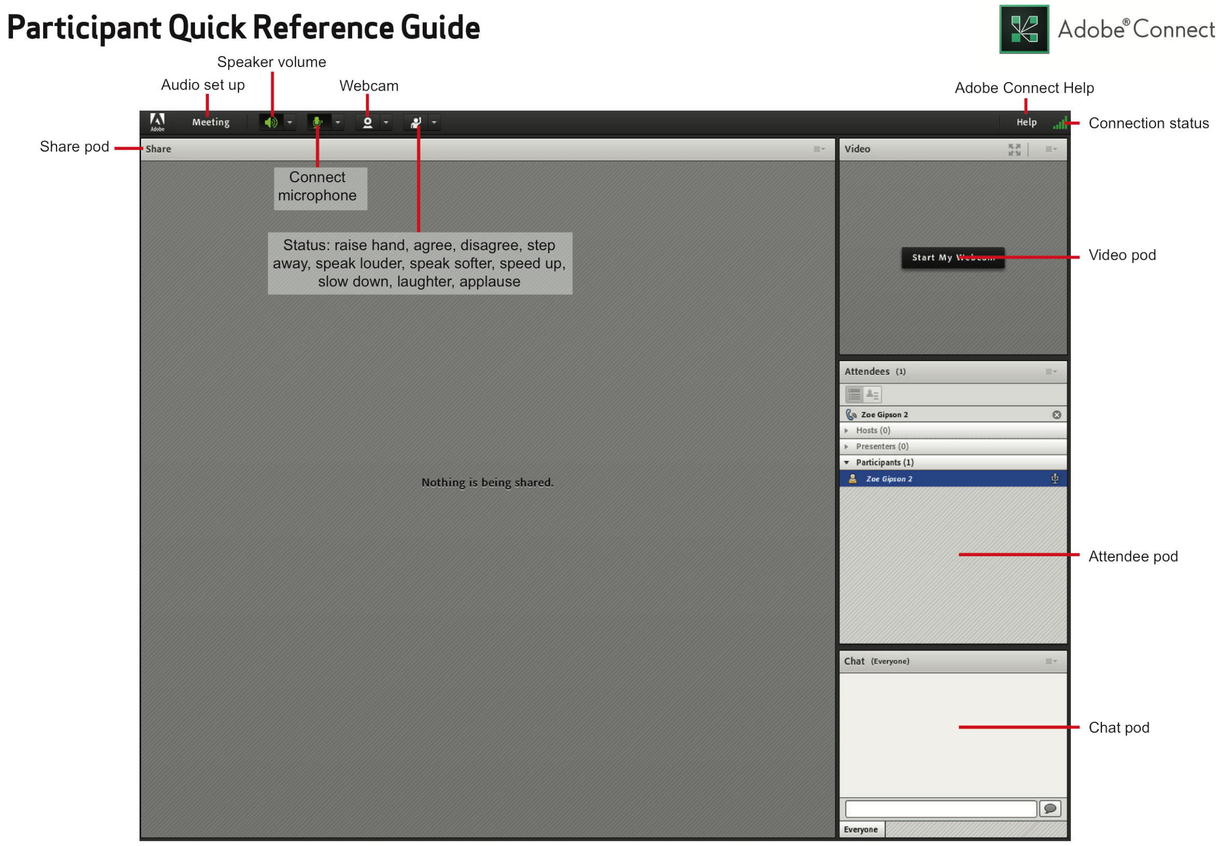
Discussion

- 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/

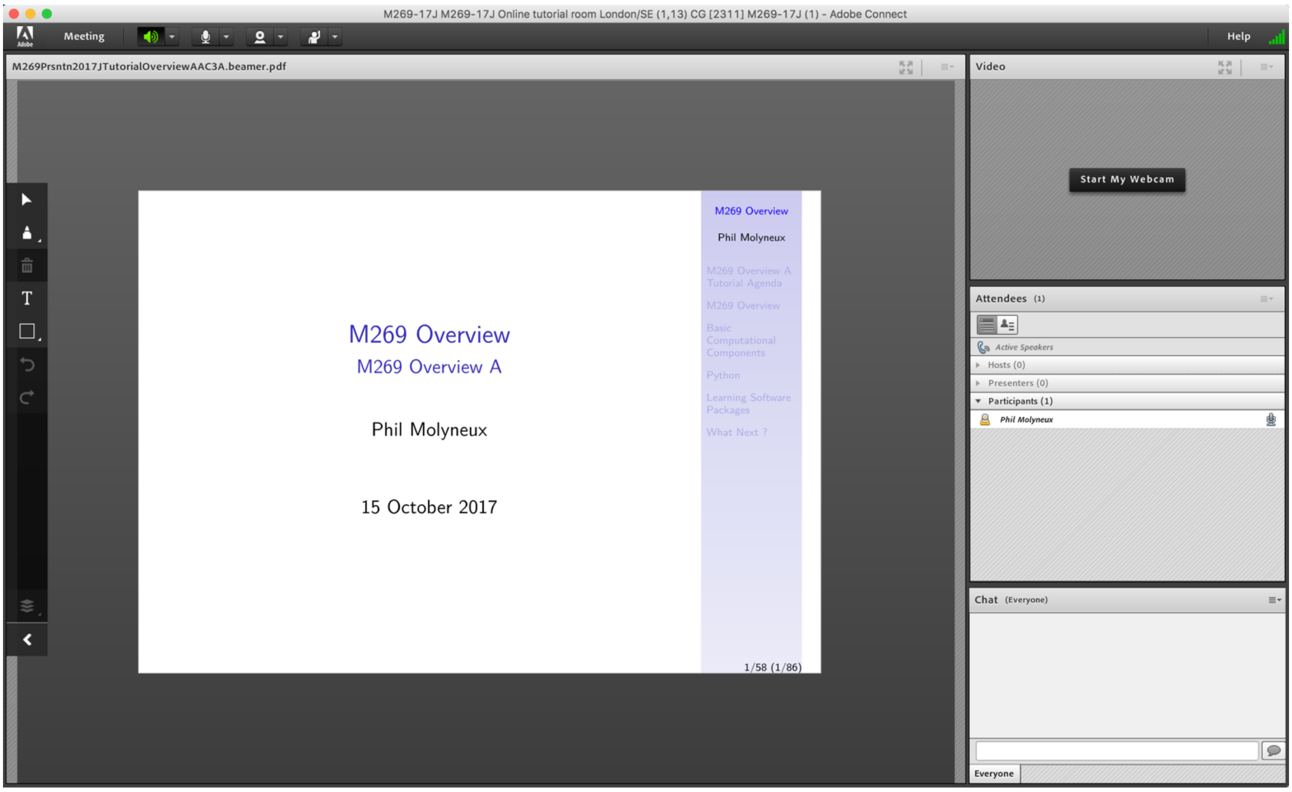
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface — Student View

Adobe Connect Interface — Student Quick Reference



Adobe Connect Interface — Student View



2.2 Adobe Connect Settings

Adobe Connect Settings

- **Everybody: Audio Settings** Meeting > Audio Setup Wizard...
- **Audio** Menu bar > Audio > Microphone rights for Participants ✔
- Do not *Enable single speaker mode*
- **Drawing Tools** Share pod menu bar > Draw (1 slide/screen)
- Share pod menu bar > Menu icon > Enable Participants to draw ✔ gray
- Meeting > Preferences > Whiteboard > Enable Participants to draw ✔
- Cancel hand tool ... Do not enable green pointer...
- Meeting > Preferences > Attendees Pod ✘ *Raise Hand notification*
- Meeting > Preferences > Display Name Display First & Last Name
- **Cursor** Meeting > Preferences > General tab > Host Cursors > Show to all attendees ✔ (default *Off*)
- Meeting > Preferences > Screen Share > Cursor > Show Application Cursor
- **Webcam** Menu bar > Webcam > Enable Webcam for Participants ✔
- **Recording** Meeting > Record Meeting... ✔

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**

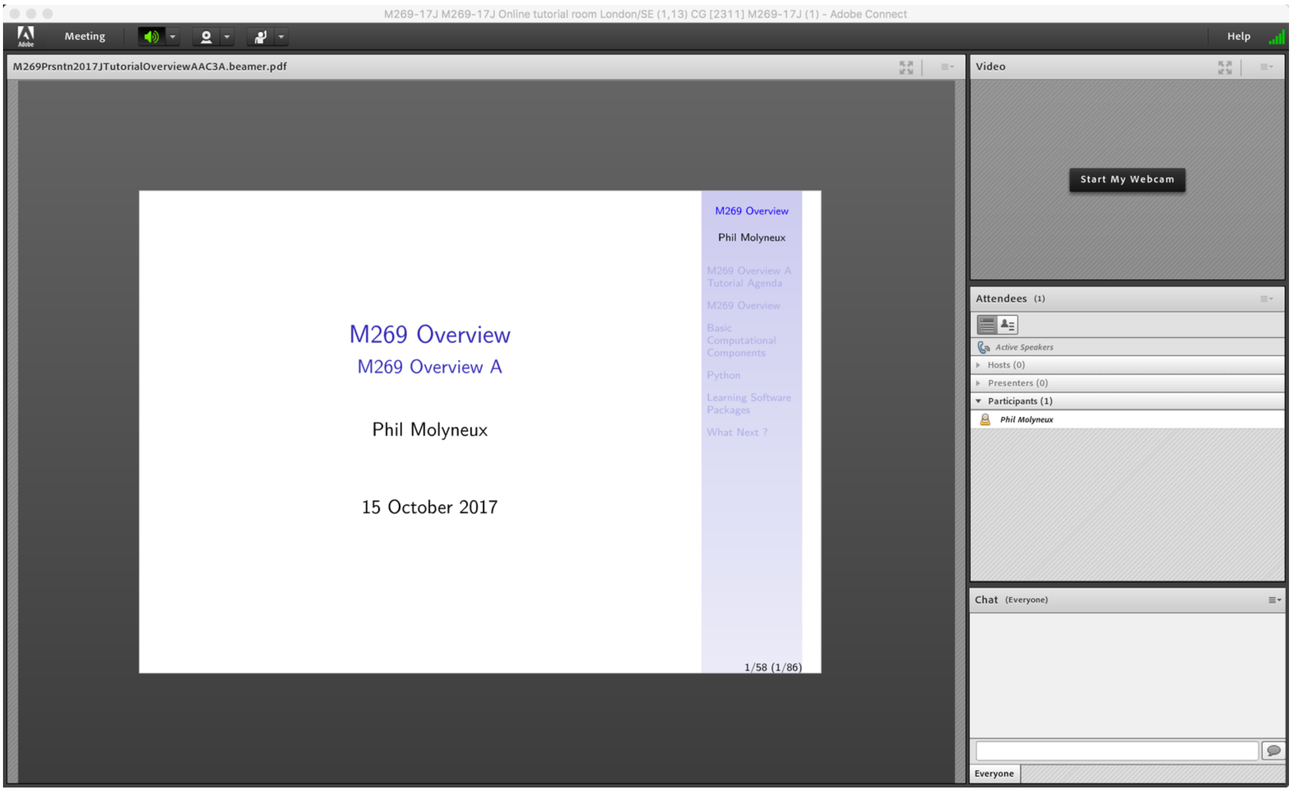
  Meeting > Enable/Disable Presenter Only Area

Adobe Connect — Keystroke Shortcuts

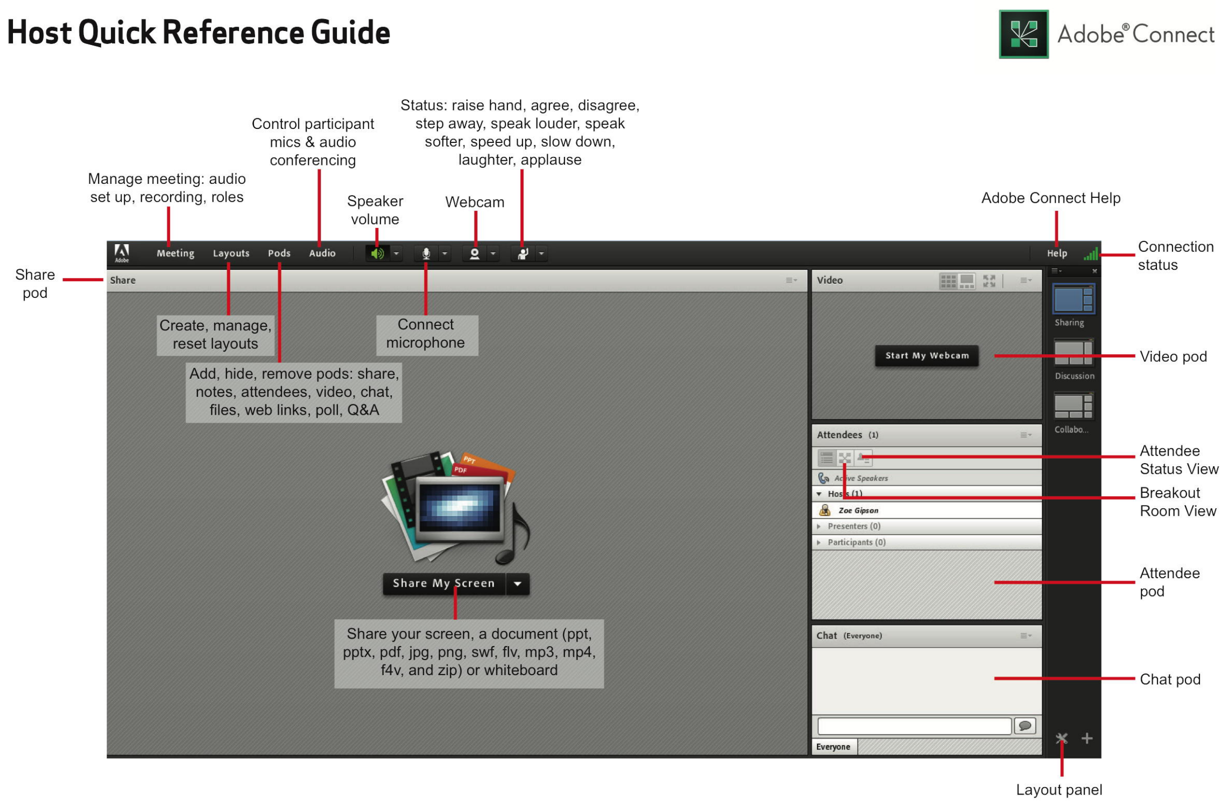
- Keyboard shortcuts in Adobe Connect
- **Toggle Mic** ⌘+M (Mac), Ctrl+M (Win) (On/Disconnect)
- **Toggle Raise-Hand status** ⌘+E
- **Close dialog box** ⎋ (Mac), Esc (Win)
- **End meeting** ⌘+\

2.3 Adobe Connect Interface — Student & Tutor Views

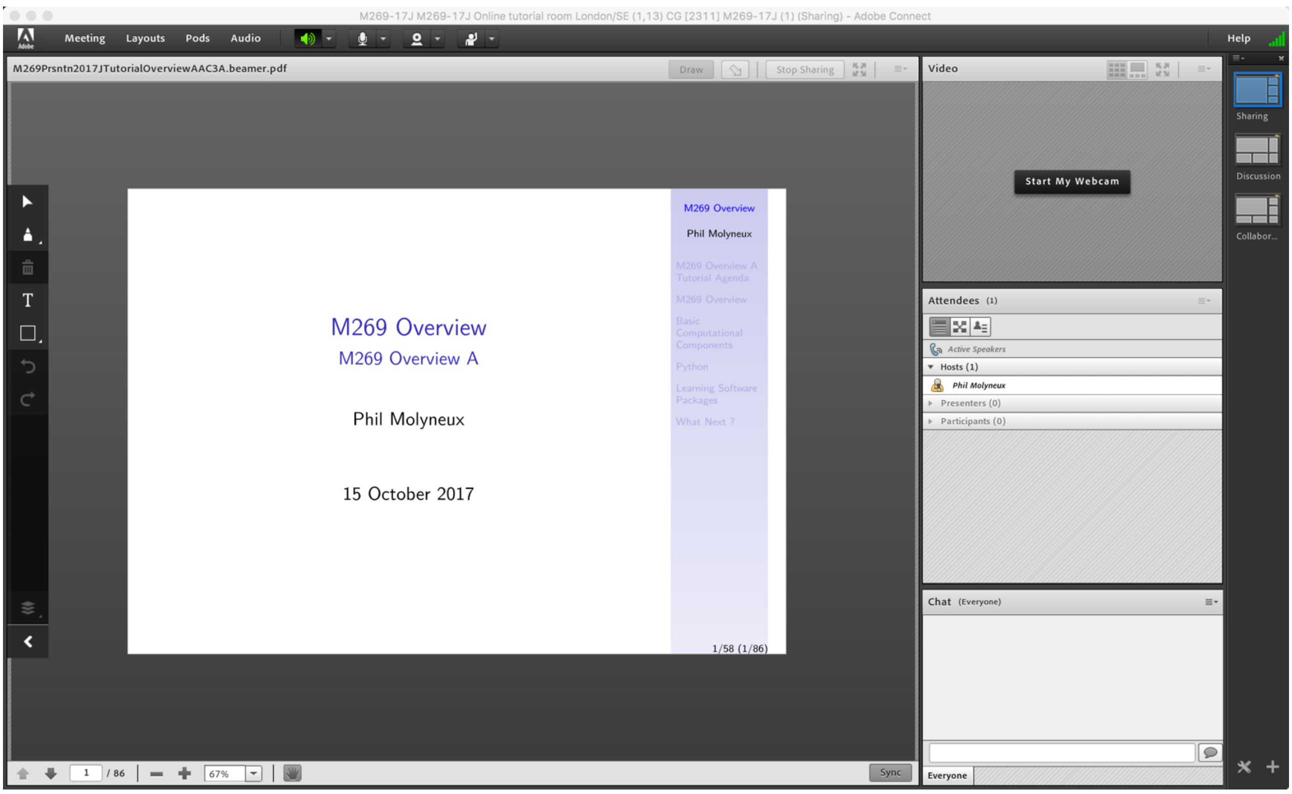
Adobe Connect Interface — Student View (default)



Adobe Connect Interface — Tutor Quick Reference



Adobe Connect Interface — Tutor View



2.4 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

2.5 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.
- **Attendance Report** see course Web site for joining room

2.6 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

2.7 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
- 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)

2.8 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

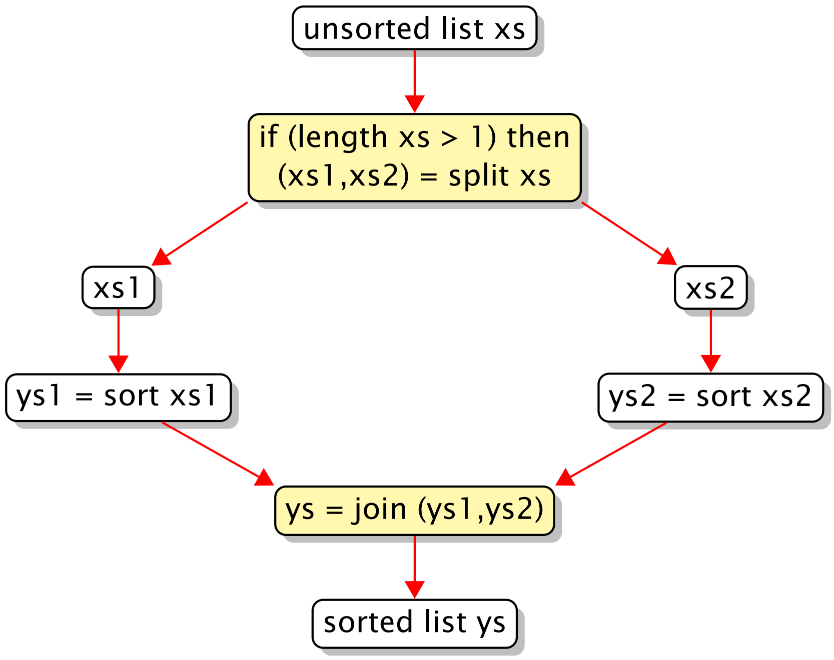
Go to Table of Contents

3 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



3.1 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*
  - Timsort by Tim Peters — used in Python and Java — combination of merge and insertion sorts
  - Haskell — modified Mergesort by Ian Lynagh in GHC implementation

4 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.

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

Iterative Factorial

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

def accProd(n,x) :
  if n == 1 :
    return x
  else :
    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

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

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

```python
def facWhile(n) :
  x = 1

  while n > 1 :
    x = n * x
    n = (n - 1)

  return x
```

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

```python
def facFor(n) :
  x = 1

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

  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 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.

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

5 Some Split/Join Sorting Algorithms

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

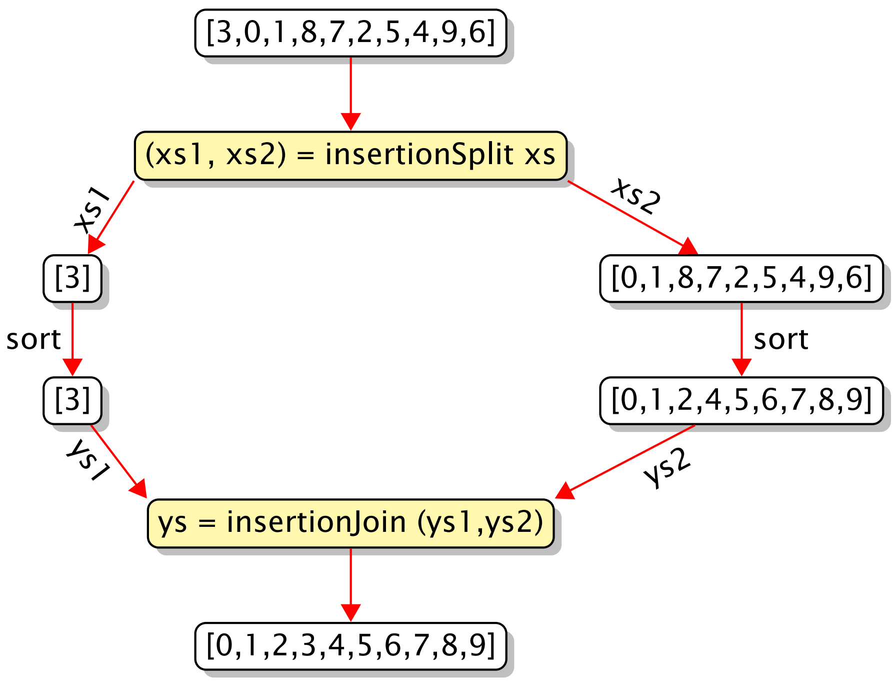
- Implementations in Python, recursive and non-recursive
- Implementations in Haskell — for comparison, optional
- Sorting via data structure — Treesort, Heap Sort

5.1 Insertion Sort

5.1.1 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



5.1.2 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

```python
def insSort01(xs) :
  if len(xs) <= 1 :
    return xs
  else :
    (xs1,xs2) = insertionSplit(xs)
    ys1 = insSort01(xs1)
    ys2 = insSort01(xs2)
    ys = insertionJoin(ys1,ys2)
    return ys

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

def insertionJoin(ys1,ys2) :
  if ys2 == [] :
    return ys1
  elif ys1[0] <= ys2[0] :
    return ys1 + ys2
  else :
    return ys2[0:1] + insertionJoin(ys1,ys2[1:])
```

5.1.3 Insertion Sort — Haskell

```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`
6. We use the `sort` function from `Data.List`.
7. The `Maybe` datatype from `Data.Maybe` will be used at the end of this script to implement the trees with data.

```haskell
  insSort  []  = []
  insSort  [x] = [x]
  insSort (x : xs)  =  ins x (insSort xs)

  ins  x  []  =  [x]
  ins  x  (y:ys)
    =  if x <= y
       then x:y:ys
       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)
- 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

5.1.4 Activity 2 — Insertion Sort: Trace an Evaluation

Insertion Sort — Haskell 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 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.

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.

5.1.5 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.

```python
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
```

5.1.6 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

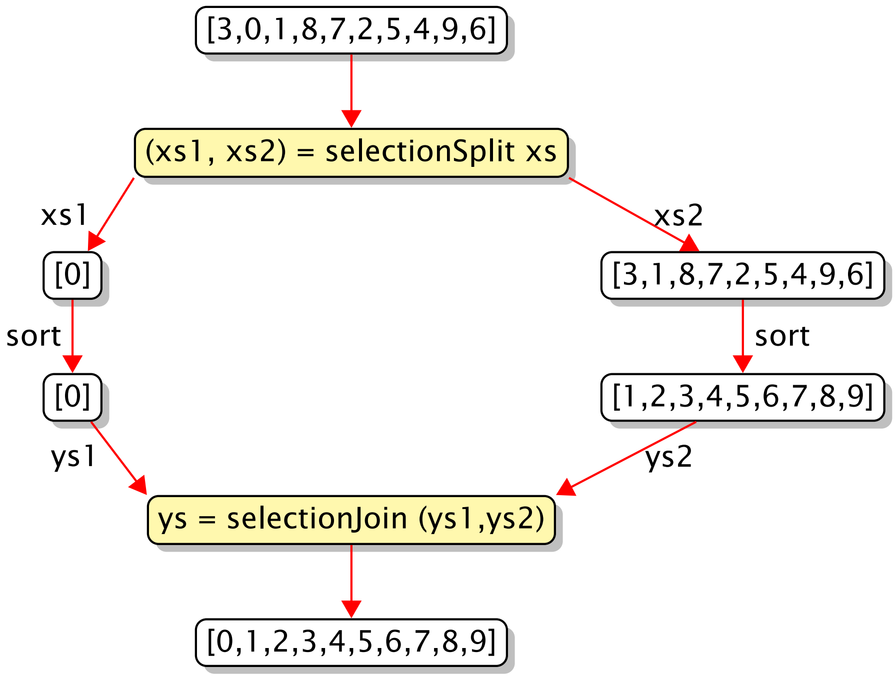
3	0	1	8	7	index = 1
0	3	1	8	7	index = 2
0	1	3	8	7	index = 3
0	1	3	8	7	index = 4
0	1	3	7	8	end

5.2 Selection Sort

5.2.1 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



5.2.2 Selection Sort — Haskell

```
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

5.2.3 Activity 4 — Selection Sort: Trace an Evaluation

Selection Sort — Haskell Recursive

- 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.

5.2.4 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))` ?

5.2.5 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.

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

    temp = xs[fillSlot]
    xs[fillSlot] = xs[maxIndex]
    xs[maxIndex] = temp
```

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

5.2.6 Activity 5 — Finding the Non-Recursive Algorithm

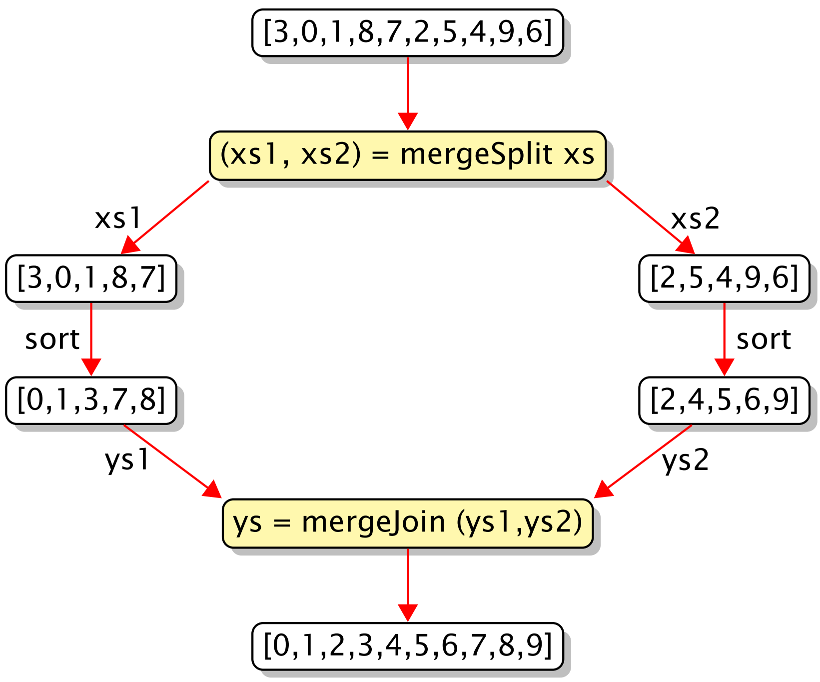
- For *Insertion Sort* and *Selection Sort* discuss how the non-recursive case can be found by considering the recursive case and doing the algorithm in place.

5.3 Merge Sort

5.3.1 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



5.3.2 Merge Sort — Haskell

```
mergeSort  []  =  []
mergeSort  [x]  =  [x]
mergeSort  xs
  =  mergeJoin  (mergeSort as) (mergeSort bs)
     where
     (as,bs) = mergeSplit xs

mergeSplit = mergeSplit2

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

mergeJoin  []  ys  =  ys
mergeJoin  xs  []  =  xs
mergeJoin  (x:xs)  (y:ys)
  |  x <= y      =  x : mergeJoin xs (y:ys)
  |  otherwise   =  y : mergeJoin (x:xs) ys
```

Haskell 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

```
mergeSplit1  []   =  ([],[])
mergeSplit1  [x]  =  ([x],[])
mergeSplit1  (x:y:zs)
  =  (x:xs, y:ys)
     where
     (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

5.3.3 Merge Sort — 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:])
```

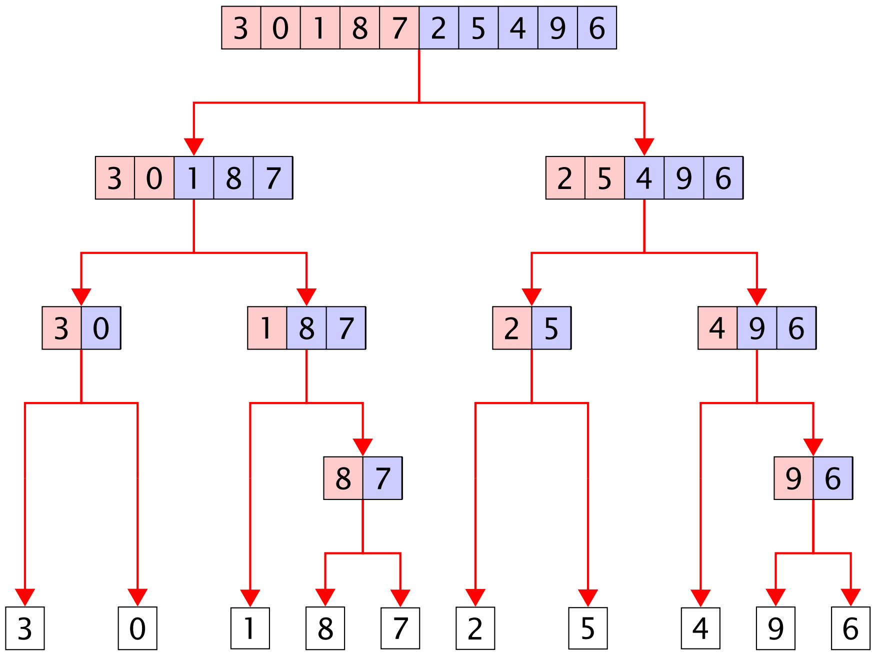
Python `mergeSplit1`

```
def mergeSplit1(xs) :
  if len(xs) == 0 :
    return  ([],[])
  elif len(xs) == 1 :
    return (xs,[])
```

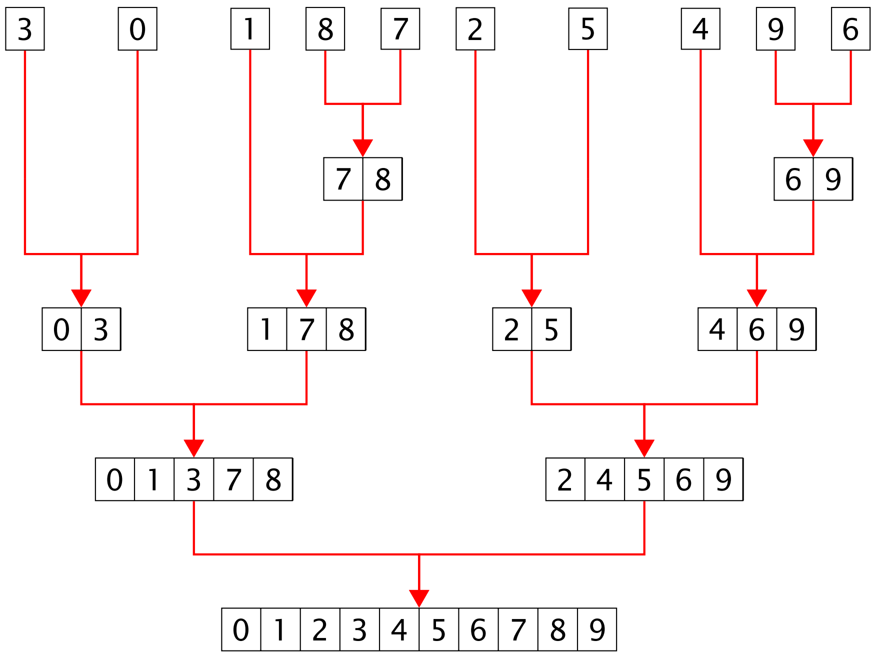
```python
    else :
        (aList,bList) = mergeSplit1(xs[2:])
        return ([xs[0]] + aList, [xs[1]] + bList)
```

5.3.4 Merge Sort Diagram

Merge Sort Split Phase



Merge Sort Join Phase



5.3.5 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)

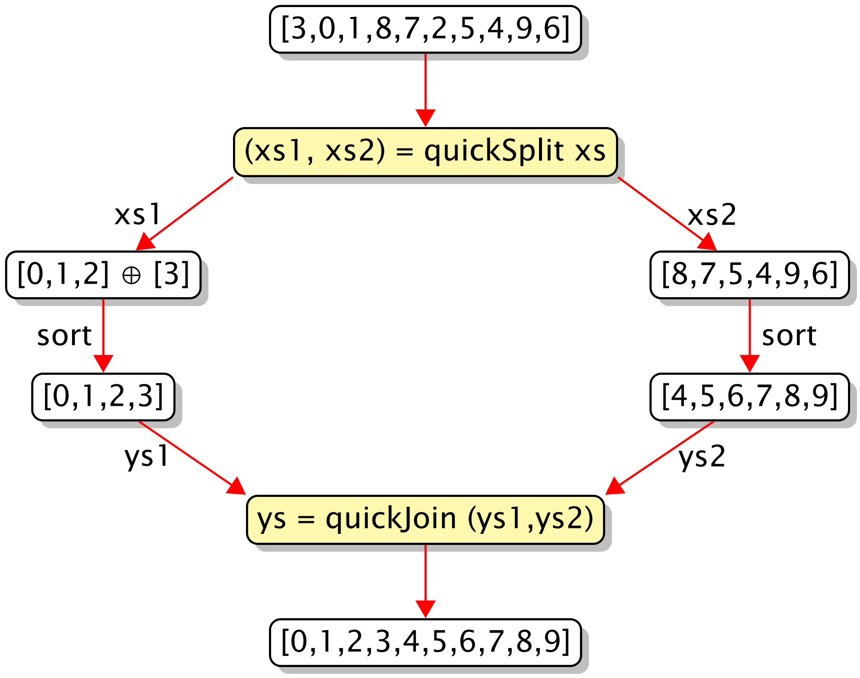
```
Python3>>> from SortingPython import *
Python3>>> xs = [3,0,1,8,7,2,5,4,9,6]
Python3>>> xsc = xs[:]
Python3>>> mergeSortInPlace(xsc)
Splitting  [3, 0, 1, 8, 7, 2, 5, 4, 9, 6]
Splitting  [3, 0, 1, 8, 7]
Splitting  [3, 0]
Singleton  [3]
Merged  [3]
Singleton  [0]
Merged  [0]
Merging  [3] , [0] to [0,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

5.4 Quicksort

5.4.1 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



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

5.4.2 Quicksort — Haskell

```
quickSort1  []  =  []
quickSort1  (x:xs)
  =  quickSort1 [y | y <- xs, y < x]
     ++ [x]
     ++ 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

5.4.3 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

5.4.4 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 shows that Python is weakly typed (and the author of this code comes from JavaScript)

5.4.5 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`

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

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

    splitPoint = partition(xs,fst,lst)

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

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

```
  while not done :
    while leftMk <= rightMk and \
            xs[leftMk] <= pivotValue :
      leftMk = leftMk + 1
    while xs[rightMk] >= pivotValue and \
            rightMk >= leftMk :
      rightMk = rightMk - 1

    if rightMk < leftMk :
      done = True
    else :
      xs[leftMk], xs[rightMk] = xs[rightMk], xs[leftMk]

  xs[fst], xs[rightMk] = xs[rightMk], xs[fst]
  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.

5.5 Bubble Sort

5.5.1 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.

5.5.2 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

```
bubbleSort  xs
  =  if (ts == xs) then ts else (bubble ts)
     where
     ts = bubble xs
```

```haskell
bubble []  = []
bubble [x] = [x]
bubble (x1:x2:xs)
  | x1 > x2     =  x2 : (bubble (x1 : xs))
  | otherwise   =  x1 : (bubble (x2 : xs))
```

- 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

5.5.3 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)

```python
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])

6 What Next ?

Topics in Units 3 and 4

- 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
- Following this section, there are some slides on Binary Trees and tree sort

7 Sorting via a Data Structure — Tree Sort

7.1 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

7.2 Tree Sort — Python

```
from collections import namedtuple

EmptyTreeBT = None

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

# Binary Tree Operations

def makeEmptyBT() :
  return EmptyTreeBT

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

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

- 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.
- 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.

```
def insertBST(x,t) :
  if isEmptyBT(t) :
    return NodeBT(x,EmptyTreeBT,EmptyTreeBT)
  else :
    y = t.dataBT
    if x < y :
      return NodeBT(y, insertBST(x,t.leftBT),t.rightBT)
    elif x > y :
      return NodeBT(y, t.leftBT, insertBST(x,t.rightBT))
    else :
      return t

def insertListBST(t,xs) :
```

```
  if xs == [] :
    return t
  else :
    return insertListBST(insertBST(xs[0],t),xs[1:])
```

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

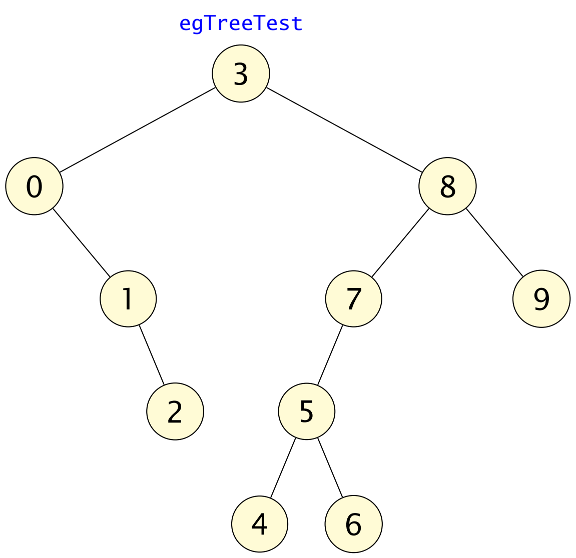
```
def inOrderBT(t) :
  if isEmptyBT(t) :
    return []
  else :
    return (inOrderBT(t.leftBT) + [t.dataBT]
            + inOrderBT(t.rightBT))
```

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

- Example list and tree

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

egTree = insertListBST(makeEmptyBT(),xs)

egTreeTest = NodeBT(3,
               NodeBT(0,
                 EmptyTreeBT,
                 NodeBT(1,
                   EmptyTreeBT,
                   NodeBT(2, EmptyTreeBT, EmptyTreeBT))),
               NodeBT(8,
                 NodeBT(7,
                   NodeBT(5,
                     NodeBT(4, EmptyTreeBT, EmptyTreeBT),
                     NodeBT(6, EmptyTreeBT, EmptyTreeBT)),
                   EmptyTreeBT),
                 NodeBT(9, EmptyTreeBT, EmptyTreeBT)))
```

7.3 Example Tree Sort



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

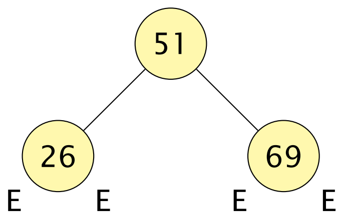
7.4 Tree Sort — Haskell

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

  ```haskell
  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):

- 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.

- Other details can be found in chapter 11 of the *Haskell Report* http://www.haskell.org/haskellwiki/Language_and_library_specification (I'm avoiding talking about ordering since you probably don't want that on trees)
- 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)

```
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
```

- The line starting insertBST :: (line 72) is a *Type Signature* which specifies the type of the function insertBST
- This is usually not required since *Haskell uses a traditional Hindley-Milner polymorphic type system to provide a static type semantics but the type system has been extended with type classes (or just classes) that provide a structured way to introduce overloaded functions.* (Marlow and Peyton Jones, 2010, section 4.1) (this is definitely not part of the M269 tutorial)
- 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).

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

treeSort :: Ord a => [a] -> [a]
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

Haskell — Alternative Definitions

- Alternative tree building bracketing from the right

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

insertBST01 x EmptyTreeBT
  =  NodeBT x EmptyTreeBT EmptyTreeBT

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

insertBST01 x (NodeBT y leftT rightT)
  | x < y    =  NodeBT y (insertBST01 x leftT) rightT
  | x > y    =  NodeBT y leftT (insertBST01 x rightT)
  | x == y   =  NodeBT y leftT rightT

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

Haskell — 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

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

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

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

The `fold` functions

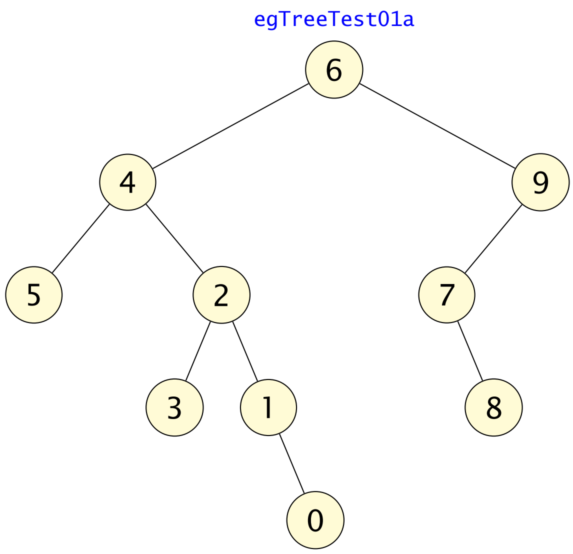
$$\begin{array}{ll} & \texttt{foldl}\ (\oplus)\ z\ [x_1,x_2,\ldots,x_n] \\ \rightarrow & (\ldots((z \oplus x_1) \oplus x_2) \oplus \ldots) \oplus x_n \\ & \texttt{foldr}\ (\oplus)\ z\ [x_1,x_2,\ldots,x_n] \\ \rightarrow & x_1 \oplus (x_2 \oplus \ldots \oplus (x_n \oplus z)\ldots) \end{array}$$

- 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

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

egTreeTesta = insertListBSTa xs

testA
  = egTreeTesta
```

```haskell
      == NodeBT 3
           (NodeBT 0
              EmptyTreeBT
              (NodeBT 1
                 EmptyTreeBT
                 (NodeBT 2 EmptyTreeBT EmptyTreeBT)))
           (NodeBT 8
              (NodeBT 7
                 (NodeBT 5
                    (NodeBT 4 EmptyTreeBT EmptyTreeBT)
                    (NodeBT 6 EmptyTreeBT EmptyTreeBT))
                 EmptyTreeBT)
              (NodeBT 9 EmptyTreeBT EmptyTreeBT))
```

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

  egTreeTest01a = insertListBST01a xs

  test01A
    = egTreeTest01a
      == NodeBT 6
           (NodeBT 4
              (NodeBT 2
                 (NodeBT 1
                    (NodeBT 0 EmptyTreeBT EmptyTreeBT)
                    EmptyTreeBT)
                 (NodeBT 3 EmptyTreeBT EmptyTreeBT))
              (NodeBT 5 EmptyTreeBT EmptyTreeBT))
           (NodeBT 9
              (NodeBT 7
                 EmptyTreeBT
                 (NodeBT 8 EmptyTreeBT EmptyTreeBT))
              EmptyTreeBT)
```



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

8 Sorting via a Data Structure — Heap Sort

8.1 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

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** `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!

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)

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!** http://learnyouahaskell.com — 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
- **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
- **Typeclassopedia** https://wiki.haskell.org/Typeclassopedia — a more formal introduction to Functors, Applicatives and Monads
- **Haskell Wikibook** https://en.wikibooks.org/wiki/Haskell

9.4 Demonstration 2 Sorting Algorithms as Dances

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

References

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

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

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

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

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

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

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.

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

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.

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 http://books.google.co.uk/books?id=sXa_mwEACAAJ.

Landin, Peter J. (1966). The next 700 programming languages. *Communications of the Association for Computing Machinery*, 9:157–166.

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.

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

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

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.

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.

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

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.

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

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.

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

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