M269 Python, Logic, ADTs

M269 Python, ADTs Prsntn2021J

Contents

1 Agenda

- Introductions
- Programming — Paradigms and Step-by-Step Guide
- Programming and Python
- Complexity and Big O Notation
- . . . with a little classical logic
- Abstract Data Type examples
- Implementing Queues
- Implementing Lists in Lists
- A look towards the next topics
  - Recursive function definitions
  - Inductive data type definitions
- *Adobe Connect* — if you or I get cut off, wait till we reconnect (or send you an email)
- Time: about 1 hour
- Do ask questions or raise points.
- Slides/Notes M269Tutorial02ProgPythonADT

Introductions — Phil

- *Name* Phil Molyneux
- *Background*
  - Undergraduate: Physics and Maths (Sussex)
  - Postgraduate: Physics (Sussex), Operational Research (Brunel), Computer Science (University College, London)
  - Worked in Operational Research, Business IT, Web technologies, Functional Programming
- *First programming languages* Fortran, BASIC, Pascal
- *Favourite Software*
  - Haskell — pure functional programming language
  - Text editors TextMate, Sublime Text — previously Emacs
  - Word processing in LaTeX — all these slides and notes
  - Mac OS X
- *Learning style* — I read the manual before using the software

Introductions — You

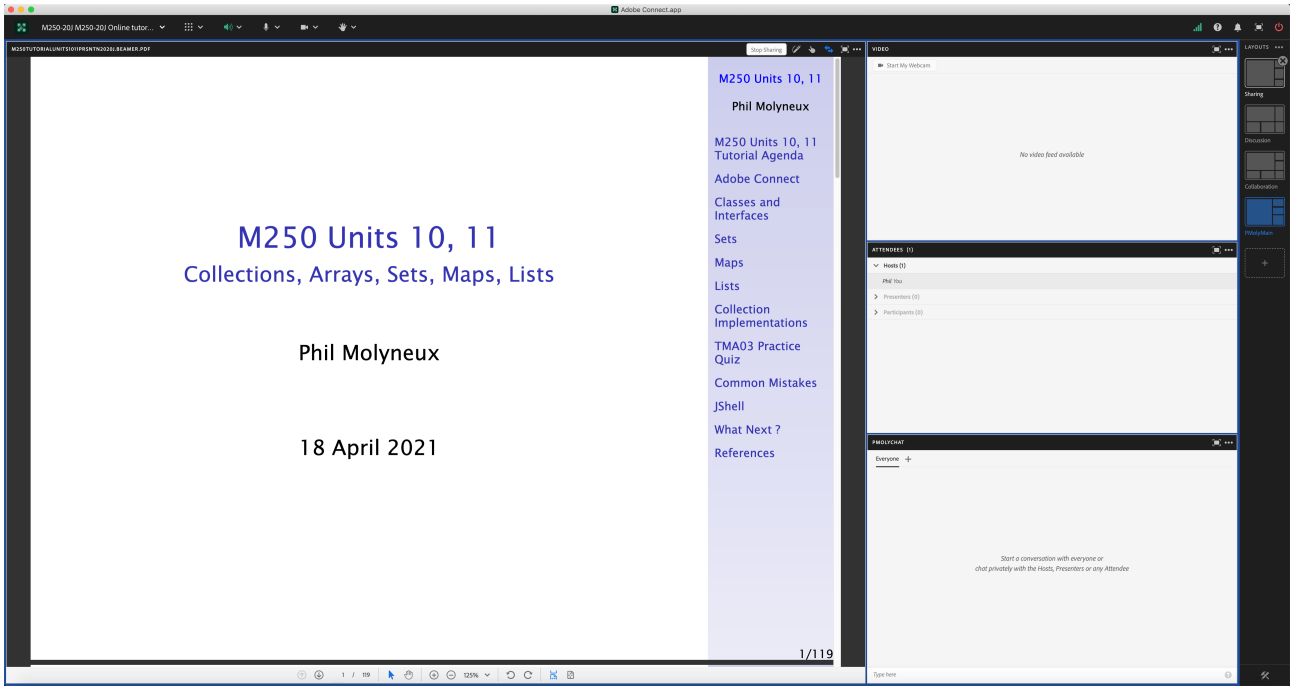
- *Name* ?
- *Favourite software/Programming language* ?
- *Favourite text editor or integrated development environment (IDE)*
- List of text editors, Comparison of text editors and Comparison of integrated development environments
- *Other OU courses* ?
- *Anything else* ?

Go to Table of Contents

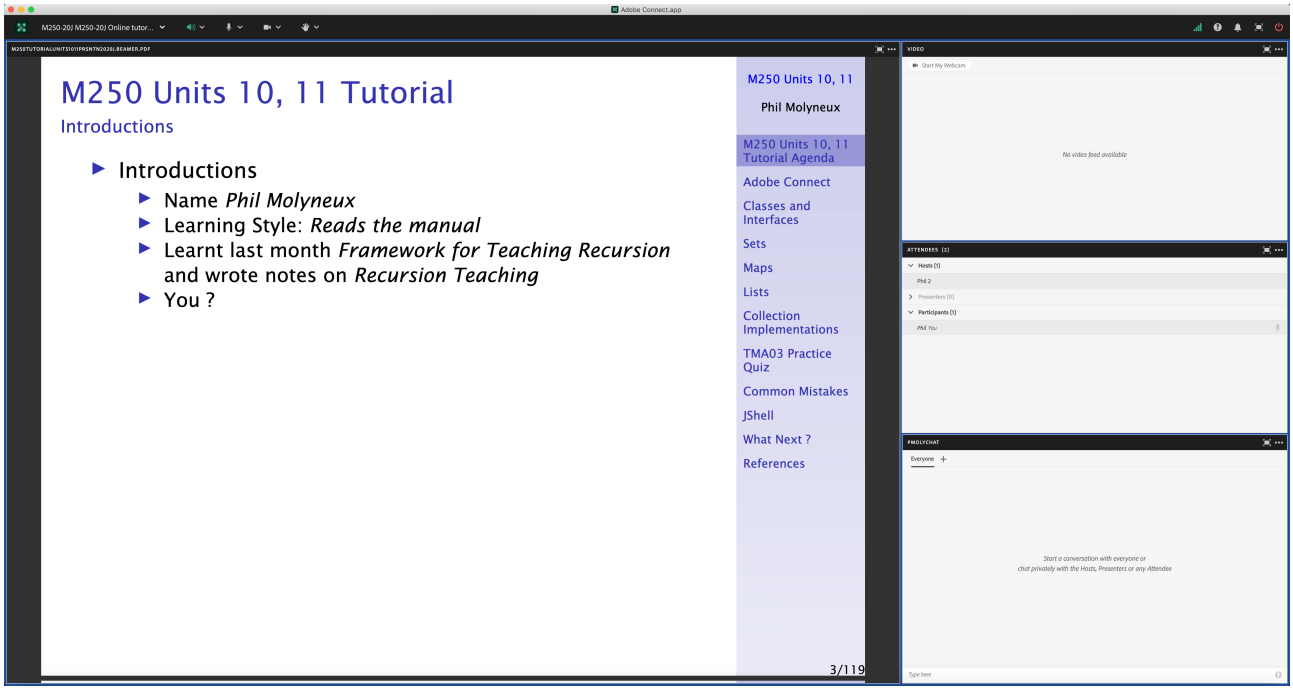
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface

Adobe Connect Interface — Host View



Adobe Connect Interface — Participant View



2.2 Adobe Connect Settings

Adobe Connect — Settings

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

Adobe Connect — Access

- **Tutor Access**

  TutorHome > M269 Website > Tutorials

  Cluster Tutorials > M269 Online tutorial room

  Tutor Groups > M269 Online tutor group room

  Module-wide Tutorials > M269 Online module-wide room

- **Attendance**

  TutorHome > Students > View your tutorial timetables

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

  Attendee Pod > Menu > Clear Everyone's Status

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

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

- **Presenter Only Area**

  Meeting > Enable/Disable Presenter Only Area

Adobe Connect — Keystroke Shortcuts

- Keyboard shortcuts in Adobe Connect
- **Toggle Mic** ⌘ + M (Mac), Ctrl + M (Win) (On/Disconnect)
- **Toggle Raise-Hand status** ⌘ + E

- **Close dialog box** ⎋ (Mac), Esc (Win)
- **End meeting** ⌘+\

2.3 Adobe Connect — Sharing Screen & Applications

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

2.4 Adobe Connect — Ending a Meeting

- *Notes for the tutor only*
- **Student:** Meeting › Exit Adobe Connect
- **Tutor:**
- **Recording** Meeting › Stop Recording ✔
- **Remove Participants** Meeting › End Meeting... ✔
  - Dialog box allows for message with default message:
  - *The host has ended this meeting. Thank you for attending.*
- **Recording availability** *In course Web site for joining the room, click on the eye icon in the list of recordings under your recording* — edit description and name
- **Meeting Information** Meeting › Manage Meeting Information — can access a range of information in Web page.
- **Delete File Upload** Meeting › Manage Meeting Information › Uploaded Content tab select file(s) and click Delete
- **Attendance Report** see course Web site for joining room

2.5 Adobe Connect — Invite Attendees

- **Provide Meeting URL** Menu › Meeting › Manage Access & Entry › Invite Participants...
- **Allow Access without Dialog** Menu › Meeting › Manage Meeting Information provides new browser window with *Meeting Information* Tab bar › Edit Information

- Check *Anyone who has the URL for the meeting can enter the room*
- Default *Only registered users and accepted guests may enter the room*
- **Reverts to default next session but URL is fixed**
- Guests have blue icon top, registered participants have yellow icon top — same icon if URL is open
- See Start, attend, and manage Adobe Connect meetings and sessions

2.6 Layouts

- **Creating new layouts** example *Sharing* layout
- Menu > Layouts > Create New Layout... > Create a New Layout dialog > Create a new blank layout and name it *PMolyMain*
- New layout has no Pods but does have Layouts Bar open (see Layouts menu)
- **Pods**
- Menu > Pods > Share > Add New Share and resize/position — initial name is *Share n*
- **Rename 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)

2.7 Chat Pods

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

2.8 Graphics Conversion for Web

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

Go to Table of Contents

3 Programming — Computational Components

3.1 Computational Components

Computational Components — Imperative

Imperative or procedural programming has statements which can manipulate global memory, have explicit control flow and can be organised into procedures (or functions)

- **Sequence** of statements

```
stmnt ; stmnt
```

- **Iteration** to repeat statements

```
while expr :
  suite

for targetList in exprList :
  suite
```

- **Selection** choosing between statements

```
if expr : suite
elif expr : suite
else : suite
```

Functional programming treats computation as the evaluation of expressions and the definition of functions (in the mathematical sense)

- **Function composition** to combine the application of two or more functions — like sequence but from right to left (notation accident of history)

```
(f . g) x = f (g x)
```

- **Recursion** — function definition defined in terms of calls to itself (with *smaller* arguments) and base case(s) which do not call itself.
- **Conditional expressions** choosing between alternatives expressions

```
if expr then expr else expr
```

3.2 Computation, Programming, Programming Languages

- M269 is not a programming course but . . .
- The course uses Python to illustrate various algorithms and data structures
- The final unit addresses the question:
- *What is an algorithm* ? What is programming ? What is a programming language ?
- So it *is* a programming course (sort of)

3.3 Example Algorithm Design

Searching

- Given an ordered list (xs) and a value (val), return
  - Position of val in xs or
  - Some indication if val is not present
- *Simple strategy:* check each value in the list in turn
- *Better strategy:* use the ordered property of the list to reduce the range of the list to be searched each turn
  - Set a range of the list
  - If val equals the mid point of the list, return the mid point
  - Otherwise half the range to search
  - If the range becomes negative, report not present (return some distinguished value)

Binary Search Iterative

```
1  def binarySearchIter(xs,val):
2    lo = 0
3    hi = len(xs) - 1

5    while lo <= hi:
6      mid = (lo + hi) // 2
7      guess = xs[mid]

9      if val == guess:
10       return mid
11     elif val < guess:
12       hi = mid - 1
13     else:
14       lo = mid + 1

16   return None
```

Binary Search Recursive

```
17 def binarySearchRec(xs,val,lo=0,hi=-1):
18   if (hi == -1):
19     hi = len(xs) - 1

21   mid = (lo + hi) // 2
```

```
23    if hi < lo:
24      return None
25    else:
26      guess = xs[mid]
27      if val == guess:
28        return mid
29      elif val < guess:
30        return binarySearchRec(xs,val,lo,mid-1)
31      else:
32        return binarySearchRec(xs,val,mid+1,hi)
```

3.4 Binary Search — Exercise

Given the *Python* definition of `binarySearchRec` from above, trace an evaluation of `binarySearchRec(xs, 25)` where `xs` is

```
xs = [2,5,7,15,17,19,21,24,27,31,37,48,57,87,95]
```

Binary Search — Solution

```
xs = [2,5,7,15,17,19,21,24,27,31,37,48,57,87,95]
binarySearchRec(xs, 25)
xs = [2,5,7,15,17,19,21,24,27,31,37,48,57,87,95]
binarySearchRec(xs,25,8,14) by line 31
xs = [2,5,7,15,17,19,21,24,27,31,37,48,57,87,95]
binarySearchRec(xs,25,8,10) by line 31
xs = [2,5,7,15,17,19,21,24,27,31,37,48,57,87,95]
binarySearchRec(xs,25,8,8) by line 29
xs = [2,5,7,15,17,19,21,24,27,31,37,48,57,87,95]
binarySearchRec(xs,25,8,7) by line 29
Return value: None by line 23
```

3.5 Binary Search — Comparison

- Both forms compare the given value (`val`) to the mid-point value of the range of the list (`xs[mid]`)
- If not found, the range is adjusted via assignment in a `while` loop (iterative) or function call (recursive)
- The recursive version has *default parameter* values to initialise the function call (evil, should be a helper function)
- There are two base cases:
  - The value is found (`val == guess`)
  - The range becomes negative (`hi < lo`)
- The return value is either `mid` or `None`
- What is the *type* of the binary search function ?

Binary Search — Performance

- *Linear search* — number of comparisons
  - *Best case* 1 (first item in the list)
  - *Worst case* n (last item)
  - *Average case* $\frac{1}{2}n$
- *Binary search* — number of comparisons
  - *Best case* 1 (middle item in the list)
  - *Worst case* $\log_2 n$ (steps to see all)
  - *Average case* $\log_2 n - 1$ (steps to see half)

3.6 Writing Programs & Thinking

The Steps

1. Invent a *name* for the program (or function)
2. What is the *type* of the function ? What sort of *input* does it take and what sort of *output* does it produce ? In Python a type is implicit; in other languages such as Haskell a type signature can be explicit.
3. Invent *names* for the input(s) to the function (*formal parameters*) — this can involve thinking about possible *patterns* or *data structures*
4. What restrictions are there on the input — state the preconditions.
5. What must be true of the output — state the postconditions.
6. *Think* of the definition of the function body.

The *Think* Step

- **How to Think**
  1. Think of an example or two — what should the program/function do ?
  2. Break the inputs into separate cases.
  3. Deal with simple cases.
  4. Think about the result — try your examples again.
- **Thinking Strategies**
  1. Don't think too much at one go — break the problem down. Top down design, step-wise refinement.
  2. What are the inputs — describe all the cases.
  3. Investigate choices. What data structures ? What algorithms ?
  4. Use common tools — bottom up synthesis.
  5. Spot common function application patterns — generalise & then specialise.

6. Look for good *glue* — to combine functions together.

4 Python

4.1 Learning Python

- Miller & Ranum Problem Solving with Algorithms and Data Structures using Python
- Python 3 Documentation
- Python Tutorial
- Python Language Reference
- Python Library Reference
- Hitchhiker's Guide to Python
- Stackoverflow on Python
- Dive into Python 3

4.2 Basic Python

Python Usage

- How do you enter an interactive Python shell ?
- How do you exit Python in *Terminal* (Mac) or *Command prompt* (Windows) ?
- How do you get help in a shell ?
- How do you exit the *interactive help utility* ?
- How do you enter an interactive Python shell ?

  Windows PythonWin Shell from *Toolbox*; *Mac python3* in *Terminal*
- How do you exit Python in *Terminal* (Mac) or *Command prompt* (Windows) ?

  `quit()`
- How do you get help in a shell ?

  `help()`
- How do you exit the *interactive help utility* ?

  `quit`

Sequences Indexing, Slices

- `xs[i:j:k]` is defined to be the sequence of items from index i to (j-1) with step k.
- If k is omitted or None, it is treated as 1.
- If i or j are negative then they are relative to the end.

- If i is omitted or None use 0.
- If j is omitted or None use `len(xs)`

Python Quiz — Lists

Given the following definitions

```
xs = [10.9,25,"Phil",3.14,42,1985]
ys = [[5]] * 3
```

Evaluate

```
xs[1]
xs[0]
xs[5]
ys
xs[1:3]
xs[::2]
xs[1:-1]
xs[-3]
xs[:]
ys[0].append(4)
```

Python Quiz — Lists — Answers

Given the following definitions

```
xs = [10.9,25,"Phil",3.14,42,1985]
ys = [[5]] * 3
```

Evaluate

```
xs[1]           == 25
xs[0]           == 10.9
xs[5]           == 1985
ys              == [[5],[5],[5]]
xs[1:3]         == [25, 'Phil']
xs[::2]         == [10.9, 'Phil', 42]
xs[1:-1]        == [25, 'Phil', 3.14, 42]
xs[-3]          == 3.14
xs[:]           == [10.9, 25, 'Phil', 3.14, 42, 1985]
ys[0].append(4) == [[5, 4], [5, 4], [5, 4]]
```

4.3 Python Workflows

Komodo Python Workflow

1. Create *someProgram*.py with assignment statements defining variables and other data along with function definitions.
2. There may be auxiliary files with other definitions (for example, *Python Activity 2.2* has Stack.py with the *Stack* class definition) — this uses the *import* statement in *someProgram*.py

   ```
   from someOtherDefinitions import someIdentifier
   ```

3. Load *someProgram*.py into *Komodo Edit* and use the *Run Python File* macro from the *Toolbox*

4. For further results, edit the file in *Komodo Edit* and and use the *Save and Run* macro from the *Toolbox*

Standalone Python Workflow

1. Create *someDefinitions*.py with assignment statements defining variables and function definitions.
2. In *Terminal* (Mac) or *Command Prompt* (Windows), navigate to *someDefinitions*.py and invoke the *Python 3* interpreter
3. Load *someDefinitions*.py into *Python 3* with one of

```
from someDefinitions import *
```

```
import someDefinitions as sdf
```

   The `as sdf` gives a shorter qualifier for the namespace — names in the file are now `sdf.x`

   Note that the commands are executed — any print statement will execute

4. At the *Python 3* interpreter prompt, evaluate expressions (may have side effects and alter definitions)

Standalone Python Workflow 2

1. For further results, edit the file in *Your Favourite Editor* and use one of the following commands:

```
reload(sdf)

import imp
imp.reload(sdf)
```

   Note the use of the name sdf as opposed to the original name.

   Read the following references about the dangers of reloading as compared to re-cycling *Python 3*

   - How to re import an updated package while in Python Interpreter?
   - How do I unload (reload) a Python module?
   - Reloading Python modules
   - How to dynamically import and reimport a file containing definition of a global variable which may change anytime

5 Complexity and Big O Notation

- Measuring program complexity introduced in section 4 of M269 Unit 2
- See also Miller and Ranum chapter 2 Big-O Notation
- See also Wikipedia: Big O notation

- See also Big-O Cheat Sheet
- Complexity of algorithm measured by using some surrogate to get rough idea
- In M269 mainly using assignment statements
- For *exact* measure we would have to have cost of each operation, knowledge of the implementation of the programming language and the operating system it runs under.
- But mainly interested in the following questions:
- **(1)** Is algorithm A more efficient than algorithm B for large inputs ?
- **(2)** Is there a lower bound on any possible algorithm for calculating this particular function ?
- **(3)** Is it always possible to find a polynomial time (n^k) algorithm for any function that is computable
- — the later questions are addressed in Unit 7

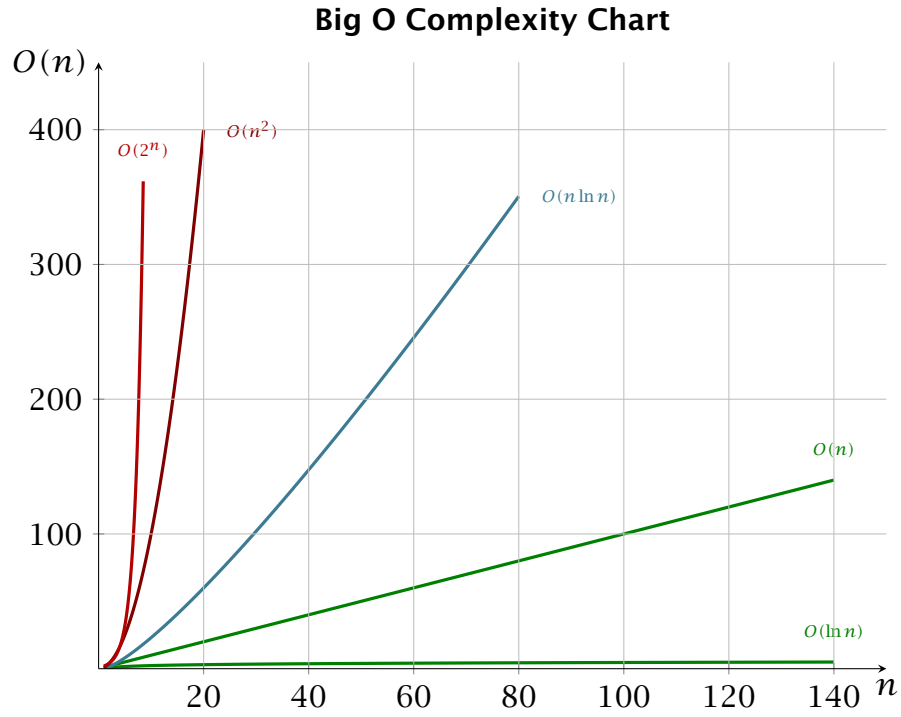
Orders of Common Functions

- $O(1)$ **constant** — look-up table
- $O(\log n)$ **logarithmic** — binary search of sorted array, binary search tree, binomial heap operations
- $O(n)$ **linear** — searching an unsorted list
- $O(n \log n)$ **loglinear** — heapsort, quicksort (best and average), merge sort
- $O(n^2)$ **quadratic** — bubble sort (worst case or naive implementation), Shell sort, quicksort (worst case), selection sort, insertion sort
- $O(n^c)$ **polynomial**
- $O(c^n)$ **exponential** — travelling salesman problem via dynamic programming, determining if two logical statements are equivalent by brute force
- $O(n!)$ **factorial** — TSP via brute force.

Tyranny of Asymptotics

- Table from Bentley (1984, page 868)
- Cubic algorithm on Cray-1 supercomputer with FORTRAN$3.0n^3$ nanoseconds
- Linear algorithm on TRS-80 micro with BASIC $19.5n \times 10^6$ nanoseconds

N	Cray-1	TRS-80
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10000	49 mins	3.2 mins
100000	35 days	32 mins
1000000	95 yrs	5.4 hrs



Big O Notation

- Abuse of notation — we write $f(x) = O(g(x))$
- but $O(g(x))$ is the class of all functions $h(x)$ such that $|h(x)| \leqslant C|g(x)|$ for some constant C
- So we should write $f(x) \in O(g(x))$ (but we don't)
- We ought to use a notation that says that (informally) the function f is bounded both above and below by g asymptotically
- This would mean that for big enough x we have

  $k_1 g(x) \leqslant f(x) \leqslant k_2 g(x)$ for some k_1, k_2
- This is Big Theta, $f(x) = \Theta(g(x))$
- But we use Big O to indicate an asymptotically tight bound where Big Theta might be more appropriate
- See Wikipedia: Big O Notation
- This could be Maths phobia generated confusion

5.1 Complexity Example

```
5  def someFunction(aList) :
6    n = len(aList)
7    best = 0
8    for i in range(n) :
9      for j in range(i + 1, n + 1) :
10       s = sum(aList[i:j])
11       best = max(best, s)
12   return best
```

- Example from M269 Unit 2 page 46
- Code in file M269TutorialProgPythonADT.py
- What does the code do ?
- (It was a *famous* problem from the late 1970s/early 1980s)
- Can we construct a more efficient algorithm for the same computational problem ?
- The code calculates the maximum subsegment of a list
- Described in Bentley (1984),Bentley (1986, column 7),Bentley (2000, column 8) Also in Gries (1989)
- These are all in a procedural programming style (as in C, Java, Python)
- Problem arose from medical image processing.
- A functional approach using Haskell is in Bird (1998, page 134), Bird (2014, page 127, 133) — a variant on this called the *Not the maximum segment sum* is given in Bird (2010, Page 73) — both of these *derive* a linear time program from the (n^3) initial specification
- See Wikipedia: Maximum subarray problem
- See Rosetta Code: Greatest subsequential sum
- Here is the same program but modified to allow lists that may only have negative numbers
- The complexity $T(n)$ function will be slightly different
- but the Big O complexity will be the same

```
14 def maxSubSeg01(xs) :
15   n = len(xs)
16   maxSoFar = xs[0]
17   for i in range(1,n) :
18     for j in range(i + 1, n + 1) :
19       s = sum(xs[i:j])
20       maxSoFar = max(maxSoFar, s)
21   return maxSoFar
```

- Complexity function $T(n)$ for maxSubSeg01()
- Two initial assignments
- The outer loop will be executed $(n-1)$ times,
- Hence the inner loop is executed

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)}{2}\times n$$

- Assume sum() takes n assignments
- Hence $T(n)=2+(n+2)\times\left(\frac{(n-1)}{2}\times n\right)$

$$=2+(n+2)\times\left(\frac{n^2}{2}-\frac{n}{2}\right)$$

$$=2+\tfrac{1}{2}n^3-\tfrac{1}{2}n^2+n^2-n$$

$= \frac{1}{2}n^3 + \frac{1}{2}n^2 - n + 2$

- Hence $O(n^3)$
- **Developing a better algorithm**
- Assume we know the solution (maxSoFar) for xs[0..(i - 1)]
- We extend the solution to xs[0..i] as follows:
- The maximum segment will be either maxSoFar
- or the sum of a sublist ending at i (maxToHere) if it is bigger
- This reasoning is similar to divide and conquer in binary search or Dynamic programming (see Unit 5)
- Keep track of both maxSoFar and maxToHere — the Eureka step
- **Developing a better algorithm** maxSubSeg02()

```
27 def maxSubSeg02(xs) :
28   maxToHere = xs[0]
29   maxSoFar  = xs[0]
30   for x in xs[1:] :
31     # Invariant: maxToHere, maxSoFar OK for xs[0..(i-1)]
32     maxToHere = max(x, maxToHere + x)
33     maxSoFar  = max(maxSoFar, maxToHere)
34   return maxSoFar
```

- Complexity function $T(n) = 2 + 2n$
- Hence $O(n)$
- What if we want more information ?
- Return the (or a) segment with max sum and position in list

```
38 def maxSubSeg03(xs) :
39   maxSoFar = maxToHere = xs[0]
40   startIdx, endIdx, startMaxToHere = 0, 0, 0
41   for i, x in enumerate(xs) :
42     if maxToHere + x < x :
43       maxToHere = x
44       startMaxToHere = i
45     else :
46       maxToHere = maxToHere + x

48     if maxSoFar < maxToHere :
49       maxSoFar = maxToHere
50       startIdx, endIdx = startMaxToHere, i

52   return (maxSoFar,xs[startIdx:endIdx+1],startIdx,endIdx)
```

- **Developing a better algorithm** maxSubSeg03()
- Complexity function worst case $T(n) = 2 + 3 + (2 + 3)n$
- Hence still $O(n)$
- Note Python assignments, enumerate() and tuple
- Sample data and output

```
56 egList = [-2,1,-3,4,-1,2,1,-5,4]

58 egList01 = [-1,-1,-1]

60 egList02 = [1,2,3]
```

```
62 assert maxSubSeg03(egList) == (6, [4, -1, 2, 1], 3, 6)

64 assert maxSubSeg03(egList01) == (-1, [-1], 0, 0)

66 assert maxSubSeg03(egList02) == (7, [1, 2, 3], 0, 2)
```

5.2 Complexity & Python Data Types

Lists

Operation	Notation	Average	Amortized Worst
Get item	`x = xs[i]`	$O(1)$	$O(1)$
Set item	`xs[i] = x`	$O(1)$	$O(1)$
Append	`xs = ys + zs`	$O(1)$	$O(1)$
Copy	`xs = ys[:]`	$O(n)$	$O(n)$
Pop last	`xs.pop()`	$O(1)$	$O(1)$
Pop other	`xs.pop(i)`	$O(k)$	$O(k)$
Insert(i,x)	`xs[i:i] = [x]`	$O(n)$	$O(n)$
Delete item	`del xs[i:i+1]`	$O(n)$	$O(n)$
Get slice	`xs = ys[i:j]`	$O(k)$	$O(k)$
Set slice	`xs[i:j] = ys`	$O(k+n)$	$O(k+n)$
Delete slice	`xs[i:j] = []`	$O(n)$	$O(n)$
Member	`x in xs`	$O(n)$	
Get length	`n = len(xs)`	$O(1)$	$O(1)$
Count(x)	`n = xs.count(x)`	$O(n)$	$O(n)$

- Source https://wiki.python.org/moin/TimeComplexity
- See https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Bags

```
5  class Bag:

7    def __init__(self):
8      self.list = []

10   def add(self, item):
11     self.list.append(item)

13   def remove(self, item):
14     self.list.remove(item)

16   def contains(self, item):
17     return item in self.list

19   def count(self, item):
20     return self.list.count(item)

22   def size(self):
23     return len(self.list)

25   def __str__(self):
26     return str(self.list)
```

Information Retrieval Functions

- **Term Frequency**, `tf`, takes a string, `term`, and a Bag, `document`

returns occurrences of `term` divided by total strings in `document`

- **Inverse Document Frequency**, `idf`, takes a string, `term`, and a list of Bags, `documents`

  returns $\log(\texttt{total}/(1+\texttt{containing}))$ — `total` is total number of Bags, `containing` is the number of Bags containing `term`

- **tf-idf**, `tf_idf`, takes a string, `term`, and a list of Bags, `documents`

  returns a sequence $[r_0, r_1, \ldots, r_{n-1}]$ such that $r_i = \texttt{tf}(\texttt{term}, d_i) \times \texttt{idf}(\texttt{term}, \texttt{documents})$

6 Exponentials and Logarithms

6.1 Exponentials and Logarithms — Definitions

- **Exponential function** $y = a^x$ or $f(x) = a^x$
- $a^n = a \times a \times \cdots \times a$ (n a terms)
- **Logarithm** reverses the operation of exponentiation
- $\log_a y = x$ means $a^x = y$
- $\log_a 1 = 0$
- $\log_a a = 1$
- Method of logarithms propounded by John Napier from 1614
- Log Tables from 1617 by Henry Briggs
- Slide Rule from about 1620–1630 by William Oughtred of Cambridge
- *Logarithm* from Greek *logos* ratio, and *arithmos* number (Chanbers Dictionary 13th edition 2014)

6.2 Rules of Indices

1. $a^m \times a^n = a^{m+n}$
2. $a^m \div a^n = a^{m-n}$
3. $a^{-m} = \dfrac{1}{a^m}$
4. $a^{\frac{1}{m}} = \sqrt[m]{a}$
5. $(a^m)^n = a^{mn}$
6. $a^{\frac{n}{m}} = \sqrt[m]{a^n}$
7. $a^0 = 1$ where $a \neq 0$

- **Exercise** Justify the above rules
- What should 0^0 evaluate to ?
- See Wikipedia: Exponentiation

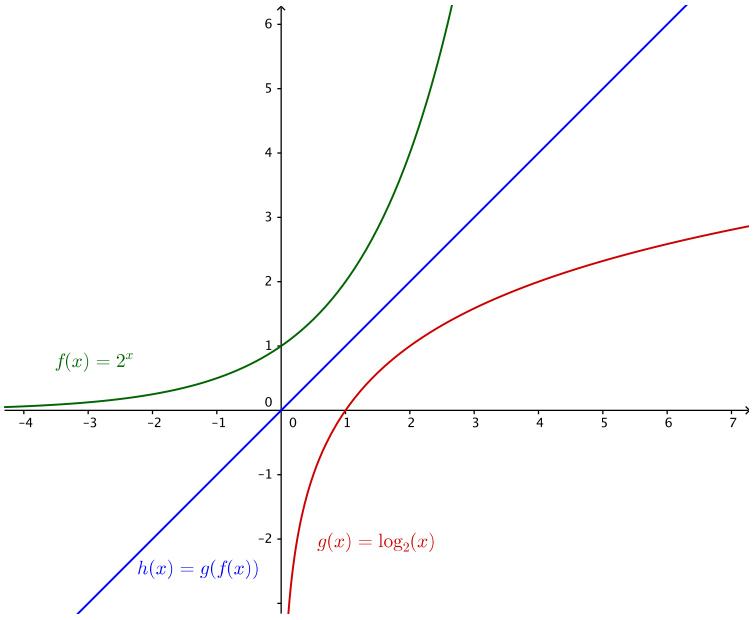
- The *justification* above probably only worked for whole or *rational* numbers — see later for exponents with *real* numbers (and the value of *logarithms*, *calculus*. . .)

6.3 Logarithms — Motivation

- Make arithmetic easier — turns multiplication and division into addition and subtraction (see later)
- Complete the range of elementary functions for differentiation and integration
- An elementary function is a function of one variable which is the composition of a finite number of arithmetic operations ($(+)$, $(-)$, $(\times)$, $(\div)$), exponentials, logarithms, constants, and solutions of algebraic equations (a generalization of nth roots).
- The elementary functions include the trigonometric and hyperbolic functions and their inverses, as they are expressible with complex exponentials and logarithms.
- See A Level FP2 for Euler's relation $\mathrm{e}^{i\theta} = \cos\theta + \mathrm{i}\sin\theta$
- In A Level C3, C4 we get $\int \frac{1}{x} = \log_{\mathrm{e}} |x| + C$
- e is Euler's number 2.71828. . .

6.4 Exponentials and Logarithms — Graphs

- See GeoGebra file expLog.ggb



6.5 Laws of Logarithms

- **Multiplication law** $\log_a xy = \log_a x + \log_a y$

- **Division law** $\log_a\left(\frac{x}{y}\right) = \log_a x - \log_a y$
- **Power law** $\log_a x^k = k \log_a x$
- **Proof of Multiplication Law**

$$
\begin{aligned}
x &= a^{\log_a x} \\
y &= a^{\log_a y} && \text{by definition of log} \\
xy &= a^{\log_a x} a^{\log_a y} \\
&= a^{\log_a x + \log_a y} && \text{by laws of indices} \\
\text{Hence } \log_a xy &= \log_a x + \log_a y && \text{by definition of log}
\end{aligned}
$$

6.6 Arithmetic and Inverses

- *Notation helps or maybe not ?*
- **Addition** $\mathrm{add}(b, x) = x + b$
- **Subtraction** $\mathrm{sub}(b, x) = x - b$
- Inverse $\mathrm{sub}(b, \mathrm{add}(b, x)) = (x + b) - b = x$
- **Multiplication** $\mathrm{mul}(b, x) = x \times b$
- **Division** $\mathrm{div}(b, x) = x \div b = \frac{x}{b} = x/b$
- Inverse $\mathrm{div}(b, \mathrm{mul}(b, x)) = (x \times b) \div b = \frac{(x \times b)}{b} = x$
- **Exponentiation** $\exp(b, x) = b^x$
- **Logarithm** $\log(b, x) = \log_b x$
- Inverse $\log(b, \exp(b, x)) = \log_b(b^x) = x$
- *What properties do the operations have that work (or not) with the notation ?*

Arithmetic Operations — Commutativity and Associativity

- **Commutativity** $x \circledast y = y \circledast x$
- **Associativity** $(x \circledast y) \circledast z = x \circledast (y \circledast z)$
- $(+)$ and $(\times)$ are *semantically* commutative and associative — so we can leave the brackets out
- $(-)$ and $(\div)$ are not
- Evaluate $(3 - (2 - 1))$ and $((3 - 2) - 1)$
- Evaluate $(3/(2/2))$ and $((3/2)/2)$
- We have the *syntactic* ideas of left (and right) associativity
- We choose $(-)$ and $(\div)$ to be left associative
- $3 - 2 - 1$ means $((3 - 2) - 1)$
- $3/2/2$ means $((3/2)/2)$

- **Operator precedence** is also a choice (remember BIDMAS or BODMAS ?)
- If in doubt, put the brackets in

Exponentials and Logarithm — Associativity

- What should 2^{3^4} mean ?
- Let $b \wedge x \equiv b^x$
- Evaluate $(2 \wedge 3) \wedge 4$ and $2 \wedge (3 \wedge 4)$
- Evaluate $c = \log_b(\log_b((b \wedge b) \wedge x))$
- Evaluate $d = \log_b(\log_b(b \wedge (b \wedge x)))$
- Beware spreadsheets Excel and LibreOffice here
- $(2^3)^4 = 2^{12}$ and $2^{3^4} = 2^{81}$
- Exponentiation is not semantically associative
- We *choose* the syntactic left or right associativity to make the syntax nicer.
- Evaluate $c = \log_b(\log_b((b \wedge b) \wedge x))$
- $c = \log_b(x \log_b(b^b)) = \log_b(x \cdot (b \log_b b)) = \log_b(x \cdot b \cdot 1)$
- Hence $c = \log_b x + \log_b b = \log_b x + 1$
- Not symmetrical (unless b and x are both 2)
- Evaluate $d = \log_b(\log_b(b \wedge (b \wedge x)))$
- $d = \log_b((b \wedge x)(\log_b b)) = \log_b((b \wedge x) \times 1)$
- Hence $d = \log_b(b \wedge x) = x(\log_b b) = x$
- Which is what we want — so exponentiation is *chosen* to be right associative

6.7 Change of Base

- **Change of base**

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Proof: Let $y = \log_a x$

$$\begin{aligned} a^y &= x \\ \log_b a^y &= \log_b x \\ y \log_b a &= \log_b x \\ y &= \frac{\log_b x}{\log_b a} \end{aligned}$$

- Given x, $\log_b x$, find the base b
  - $b = x^{\frac{1}{\log_b x}}$

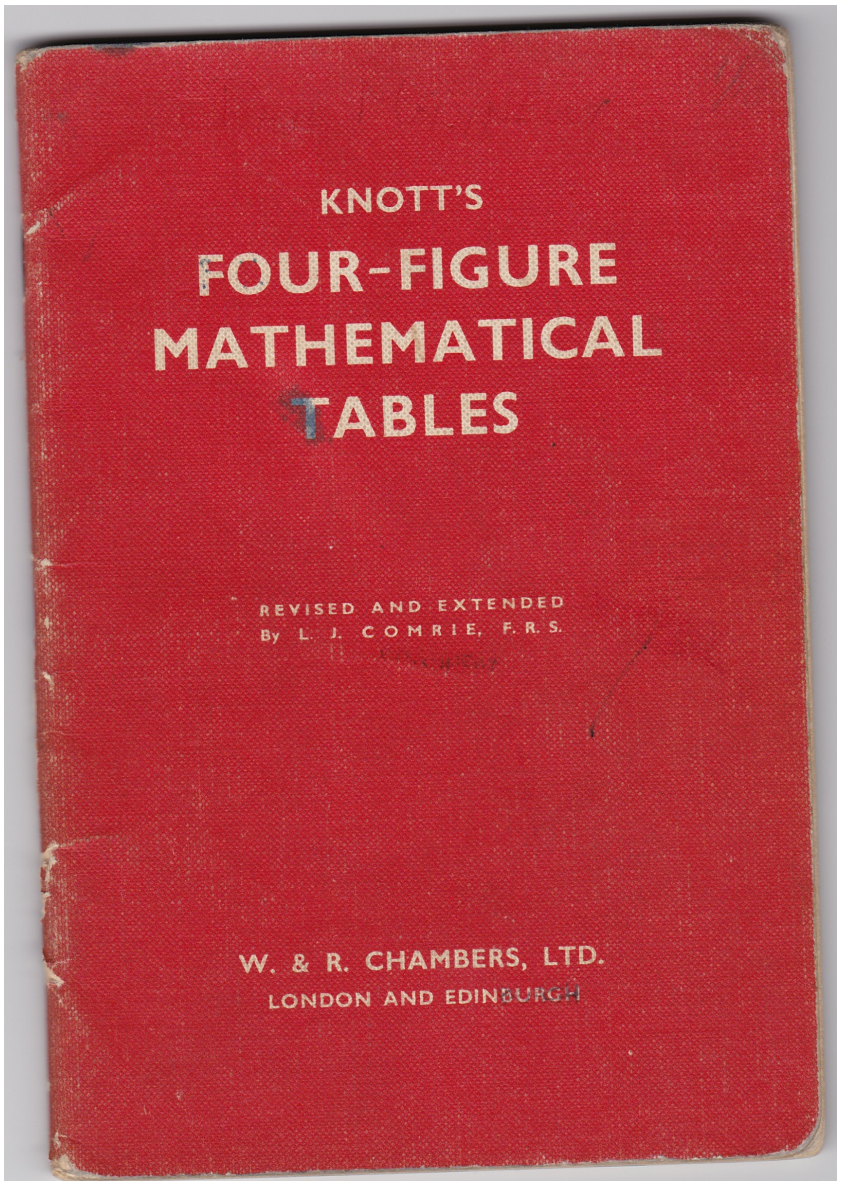
- $\log_a b = \dfrac{1}{\log_b a}$

7 Before Calculators and Computers

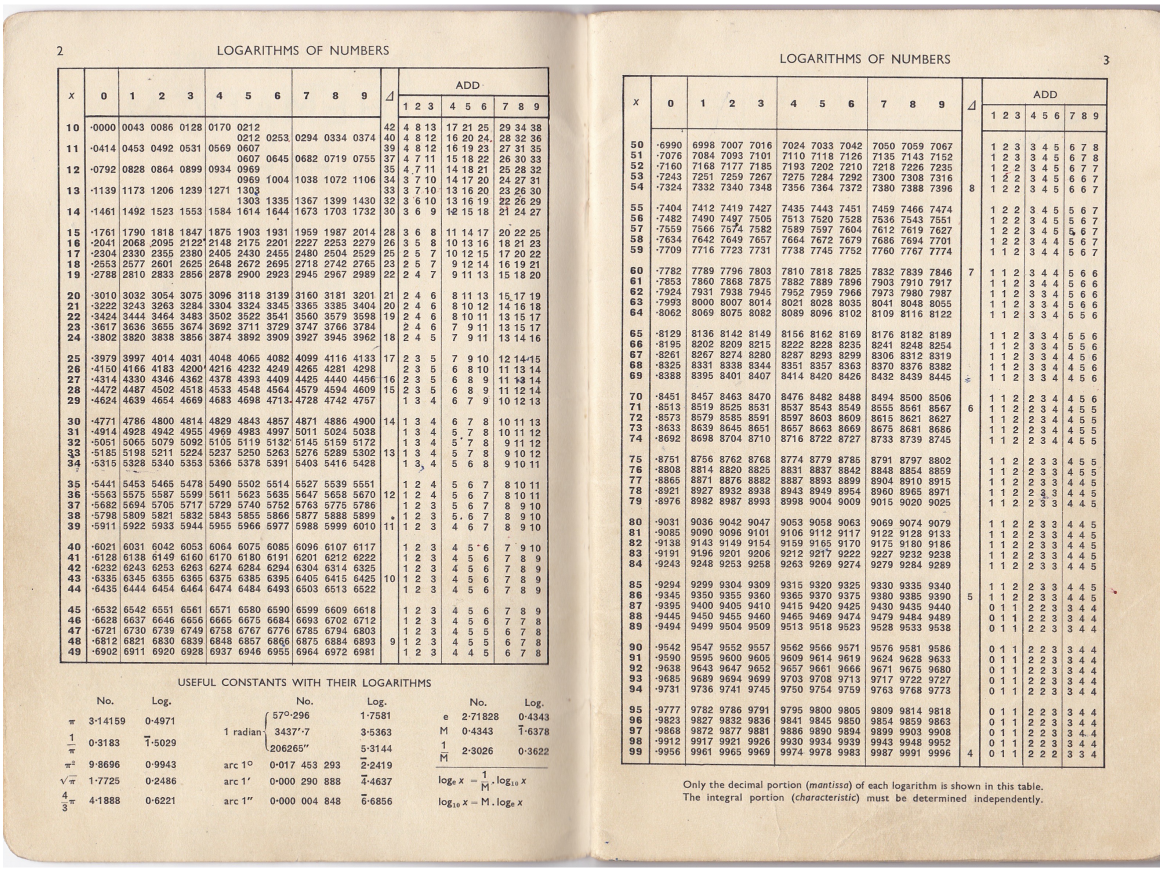
- We had computers before 1950 — they were *humans* with pencil, paper and some further aids:
- **Slide rule** invented by William Oughtred in the 1620s — major calculating tool until pocket calculators in 1970s
- **Log tables** in use from early 1600s — method of logarithms propounded by John Napier
- **Logarithm** from Greek *logos* ratio, and *arithmos* number

7.1 Log Tables

Knott's Four-Figure Mathematical Tables



Logarithms of Numbers



x	0	1	2	3	4	5	6	7	8	9	Δ	ADD								
												1	2	3	4	5	6	7	8	9
10	·0000	0043	0086	0128	0170	0212					42	4	8	13	17	21	25	29	34	38
						0212	0253	0294	0334	0374	40	4	8	12	16	20	24	28	32	36
11	·0414	0453	0492	0531	0569	0607					39	4	8	12	16	19	23	27	31	35
						0607	0645	0682	0719	0755	37	4	7	11	15	18	22	26	30	33
12	·0792	0828	0864	0899	0934	0969					35	4	7	11	14	18	21	25	28	32
						0969	1004	1038	1072	1106	34	3	7	10	14	17	20	24	27	31
13	·1139	1173	1206	1239	1271	1303					33	3	7	10	13	16	20	23	26	30
						1303	1335	1367	1399	1430	32	3	6	10	13	16	19	22	26	29
14	·1461	1492	1523	1553	1584	1614	1644	1673	1703	1732	30	3	6	9	12	15	18	21	24	27
15	·1761	1790	1818	1847	1875	1903	1931	1959	1987	2014	28	3	6	8	11	14	17	20	22	25
16	·2041	2068	2095	2122	2148	2175	2201	2227	2253	2279	26	3	5	8	10	13	16	18	21	23
17	·2304	2330	2355	2380	2405	2430	2455	2480	2504	2529	25	2	5	7	10	12	15	17	20	22
18	·2553	2577	2601	2625	2648	2672	2695	2718	2742	2765	23	2	5	7	9	12	14	16	19	21
19	·2788	2810	2833	2856	2878	2900	2923	2945	2967	2989	22	2	4	7	9	11	13	15	18	20
20	·3010	3032	3054	3075	3096	3118	3139	3160	3181	3201	21	2	4	6	8	11	13	15	17	19
21	·3222	3243	3263	3284	3304	3324	3345	3365	3385	3404	20	2	4	6	8	10	12	14	16	18
22	·3424	3444	3464	3483	3502	3522	3541	3560	3579	3598	19	2	4	6	8	10	11	13	15	17
23	·3617	3636	3655	3674	3692	3711	3729	3747	3766	3784		2	4	6	7	9	11	13	15	17
24	·3802	3820	3838	3856	3874	3892	3909	3927	3945	3962	18	2	4	5	7	9	11	13	14	16
25	·3979	3997	4014	4031	4048	4065	4082	4099	4116	4133	17	2	3	5	7	9	10	12	14	15
26	·4150	4166	4183	4200	4216	4232	4249	4265	4281	4298		2	3	5	6	8	10	11	13	14
27	·4314	4330	4346	4362	4378	4393	4409	4425	4440	4456	16	2	3	5	6	8	9	11	13	14
28	·4472	4487	4502	4518	4533	4548	4564	4579	4594	4609	15	2	3	5	6	8	9	11	12	14
29	·4624	4639	4654	4669	4683	4698	4713	4728	4742	4757		1	3	4	6	7	9	10	12	13
30	·4771	4786	4800	4814	4829	4843	4857	4871	4886	4900	14	1	3	4	6	7	8	10	11	13
31	·4914	4928	4942	4955	4969	4983	4997	5011	5024	5038		1	3	4	5	7	8	10	11	12
32	·5051	5065	5079	5092	5105	5119	5132	5145	5159	5172		1	3	4	5	7	8	9	11	12
33	·5185	5198	5211	5224	5237	5250	5263	5276	5289	5302	13	1	3	4	5	7	8	9	10	12
34	·5315	5328	5340	5353	5366	5378	5391	5403	5416	5428		1	3	4	5	6	8	9	10	11
35	·5441	5453	5465	5478	5490	5502	5514	5527	5539	5551		1	2	4	5	6	7	8	10	11
36	·5563	5575	5587	5599	5611	5623	5635	5647	5658	5670	12	1	2	4	5	6	7	8	10	11
37	·5682	5694	5705	5717	5729	5740	5752	5763	5775	5786		1	2	3	5	6	7	8	9	10
38	·5798	5809	5821	5832	5843	5855	5866	5877	5888	5899		1	2	3	5	6	7	8	9	10
39	·5911	5922	5933	5944	5955	5966	5977	5988	5999	6010	11	1	2	3	4	6	7	8	9	10
40	·6021	6031	6042	6053	6064	6075	6085	6096	6107	6117		1	2	3	4	5	6	7	9	10
41	·6128	6138	6149	6160	6170	6180	6191	6201	6212	6222		1	2	3	4	5	6	7	8	9
42	·6232	6243	6253	6263	6274	6284	6294	6304	6314	6325		1	2	3	4	5	6	7	8	9
43	·6335	6345	6355	6365	6375	6385	6395	6405	6415	6425	10	1	2	3	4	5	6	7	8	9
44	·6435	6444	6454	6464	6474	6484	6493	6503	6513	6522		1	2	3	4	5	6	7	8	9
45	·6532	6542	6551	6561	6571	6580	6590	6599	6609	6618		1	2	3	4	5	6	7	8	9
46	·6628	6637	6646	6656	6665	6675	6684	6693	6702	6712		1	2	3	4	5	6	7	7	8
47	·6721	6730	6739	6749	6758	6767	6776	6785	6794	6803		1	2	3	4	5	5	6	7	8
48	·6812	6821	6830	6839	6848	6857	6866	6875	6884	6893	9	1	2	3	4	5	5	6	7	8
49	·6902	6911	6920	6928	6937	6946	6955	6964	6972	6981		1	2	3	4	4	5	6	7	8

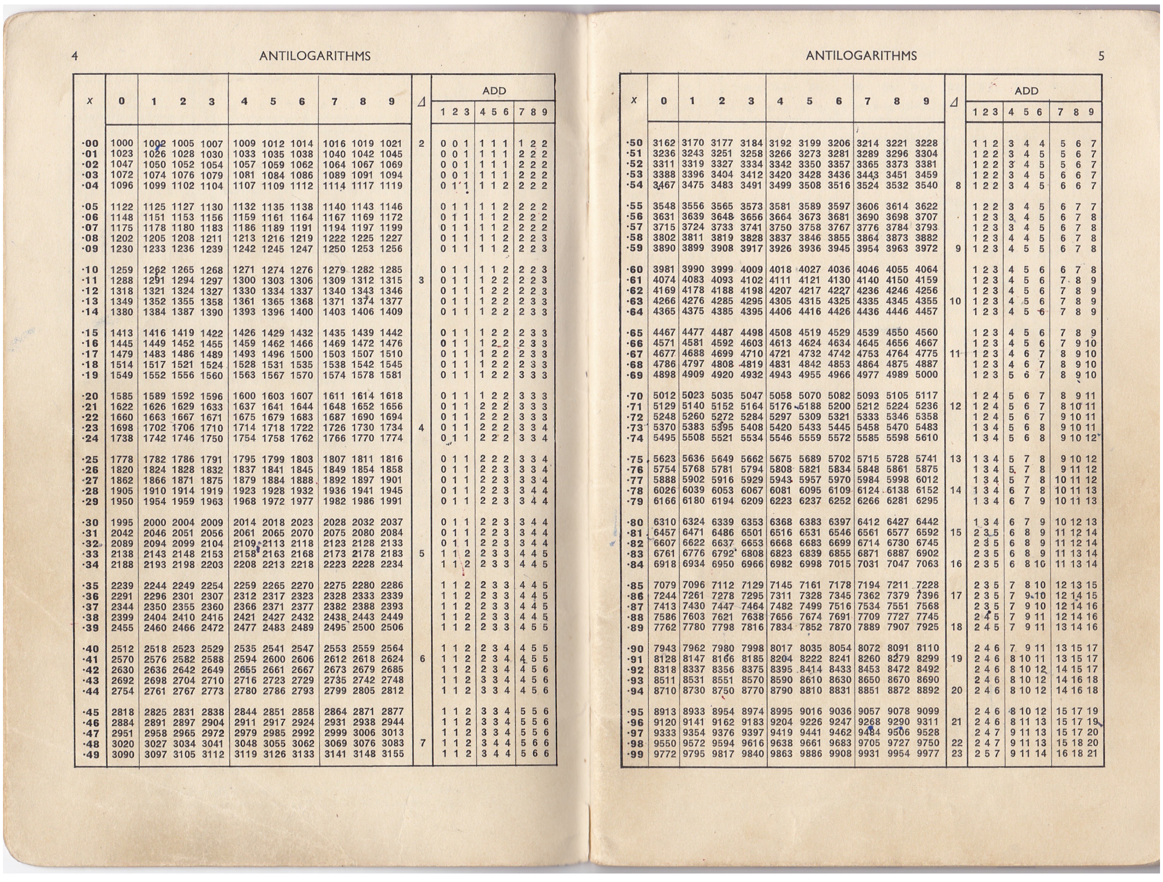
USEFUL CONSTANTS WITH THEIR LOGARITHMS

	No.	Log.		No.	Log.		No.	Log.
π	3·14159	0·4971	1 radian	57°·296	1·7581	e	2·71828	0·4343
$\frac{1}{\pi}$	0·3183	$\bar{1}$·5029		3437′·7	3·5363	M	0·4343	$\bar{1}$·6378
				206265″	5·3144	$\frac{1}{M}$	2·3026	0·3622
π^2	9·8696	0·9943	arc 1°	0·017 453 293	$\bar{2}$·2419			
$\sqrt{\pi}$	1·7725	0·2486	arc 1′	0·000 290 888	$\bar{4}$·4637	$\log_e x = \frac{1}{M} . \log_{10} x$		
$\frac{4}{3}\pi$	4·1888	0·6221	arc 1″	0·000 004 848	$\bar{6}$·6856	$\log_{10} x = M . \log_e x$		

x	0	1	2	3	4	5	6	7	8	9	Δ	ADD								
												1	2	3	4	5	6	7	8	9
50	·6990	6998	7007	7016	7024	7033	7042	7050	7059	7067		1	2	3	3	4	5	6	7	8
51	·7076	7084	7093	7101	7110	7118	7126	7135	7143	7152		1	2	3	3	4	5	6	7	8
52	·7160	7168	7177	7185	7193	7202	7210	7218	7226	7235		1	2	2	3	4	5	6	7	7
53	·7243	7251	7259	7267	7275	7284	7292	7300	7308	7316		1	2	2	3	4	5	6	6	7
54	·7324	7332	7340	7348	7356	7364	7372	7380	7388	7396	8	1	2	2	3	4	5	6	6	7
55	·7404	7412	7419	7427	7435	7443	7451	7459	7466	7474		1	2	2	3	4	5	5	6	7
56	·7482	7490	7497	7505	7513	7520	7528	7536	7543	7551		1	2	2	3	4	5	5	6	7
57	·7559	7566	7574	7582	7589	7597	7604	7612	7619	7627		1	2	2	3	4	5	5	6	7
58	·7634	7642	7649	7657	7664	7672	7679	7686	7694	7701		1	2	2	3	4	4	5	6	7
59	·7709	7716	7723	7731	7738	7745	7752	7760	7767	7774		1	1	2	3	4	4	5	6	7
60	·7782	7789	7796	7803	7810	7818	7825	7832	7839	7846	7	1	1	2	3	4	4	5	6	6
61	·7853	7860	7868	7875	7882	7889	7896	7903	7910	7917		1	1	2	3	4	4	5	6	6
62	·7924	7931	7938	7945	7952	7959	7966	7973	7980	7987		1	1	2	3	3	4	5	6	6
63	·7993	8000	8007	8014	8021	8028	8035	8041	8048	8055		1	1	2	3	3	4	5	6	6
64	·8062	8069	8075	8082	8089	8096	8102	8109	8116	8122		1	1	2	3	3	4	5	5	6
65	·8129	8136	8142	8149	8156	8162	8169	8176	8182	8189		1	1	2	3	3	4	5	5	6
66	·8195	8202	8209	8215	8222	8228	8235	8241	8248	8254		1	1	2	3	3	4	5	5	6
67	·8261	8267	8274	8280	8287	8293	8299	8306	8312	8319		1	1	2	3	3	4	4	5	6
68	·8325	8331	8338	8344	8351	8357	8363	8370	8376	8382		1	1	2	3	3	4	4	5	6
69	·8388	8395	8401	8407	8414	8420	8426	8432	8439	8445		1	1	2	3	3	4	4	5	6
70	·8451	8457	8463	8470	8476	8482	8488	8494	8500	8506		1	1	2	2	3	4	4	5	6
71	·8513	8519	8525	8531	8537	8543	8549	8555	8561	8567	6	1	1	2	2	3	4	4	5	5
72	·8573	8579	8585	8591	8597	8603	8609	8615	8621	8627		1	1	2	2	3	4	4	5	5
73	·8633	8639	8645	8651	8657	8663	8669	8675	8681	8686		1	1	2	2	3	4	4	5	5
74	·8692	8698	8704	8710	8716	8722	8727	8733	8739	8745		1	1	2	2	3	4	4	5	5
75	·8751	8756	8762	8768	8774	8779	8785	8791	8797	8802		1	1	2	2	3	3	4	5	5
76	·8808	8814	8820	8825	8831	8837	8842	8848	8854	8859		1	1	2	2	3	3	4	5	5
77	·8865	8871	8876	8882	8887	8893	8899	8904	8910	8915		1	1	2	2	3	3	4	4	5
78	·8921	8927	8932	8938	8943	8949	8954	8960	8965	8971		1	1	2	2	3	3	4	4	5
79	·8976	8982	8987	8993	8998	9004	9009	9015	9020	9025		1	1	2	2	3	3	4	4	5
80	·9031	9036	9042	9047	9053	9058	9063	9069	9074	9079		1	1	2	2	3	3	4	4	5
81	·9085	9090	9096	9101	9106	9112	9117	9122	9128	9133		1	1	2	2	3	3	4	4	5
82	·9138	9143	9149	9154	9159	9165	9170	9175	9180	9186		1	1	2	2	3	3	4	4	5
83	·9191	9196	9201	9206	9212	9217	9222	9227	9232	9238		1	1	2	2	3	3	4	4	5
84	·9243	9248	9253	9258	9263	9269	9274	9279	9284	9289		1	1	2	2	3	3	4	4	5
85	·9294	9299	9304	9309	9315	9320	9325	9330	9335	9340		1	1	2	2	3	3	4	4	5
86	·9345	9350	9355	9360	9365	9370	9375	9380	9385	9390	5	1	1	2	2	3	3	4	4	5
87	·9395	9400	9405	9410	9415	9420	9425	9430	9435	9440		0	1	1	2	2	3	3	4	4
88	·9445	9450	9455	9460	9465	9469	9474	9479	9484	9489		0	1	1	2	2	3	3	4	4
89	·9494	9499	9504	9509	9513	9518	9523	9528	9533	9538		0	1	1	2	2	3	3	4	4
90	·9542	9547	9552	9557	9562	9566	9571	9576	9581	9586		0	1	1	2	2	3	3	4	4
91	·9590	9595	9600	9605	9609	9614	9619	9624	9628	9633		0	1	1	2	2	3	3	4	4
92	·9638	9643	9647	9652	9657	9661	9666	9671	9675	9680		0	1	1	2	2	3	3	4	4
93	·9685	9689	9694	9699	9703	9708	9713	9717	9722	9727		0	1	1	2	2	3	3	4	4
94	·9731	9736	9741	9745	9750	9754	9759	9763	9768	9773		0	1	1	2	2	3	3	4	4
95	·9777	9782	9786	9791	9795	9800	9805	9809	9814	9818		0	1	1	2	2	3	3	4	4
96	·9823	9827	9832	9836	9841	9845	9850	9854	9859	9863		0	1	1	2	2	3	3	4	4
97	·9868	9872	9877	9881	9886	9890	9894	9899	9903	9908		0	1	1	2	2	3	3	4	4
98	·9912	9917	9921	9926	9930	9934	9939	9943	9948	9952		0	1	1	2	2	3	3	4	4
99	·9956	9961	9965	9969	9974	9978	9983	9987	9991	9996	4	0	1	1	2	2	2	3	3	4

Only the decimal portion (*mantissa*) of each logarithm is shown in this table. The integral portion (*characteristic*) must be determined independently.

Antilogarithms

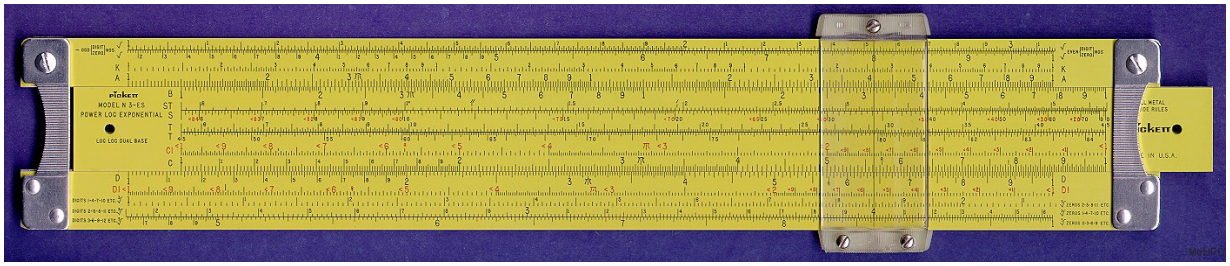


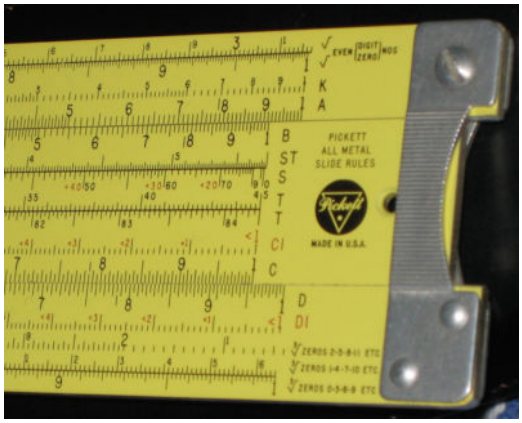
x	0	1	2	3	4	5	6	7	8	9	Δ	ADD								
												1	2	3	4	5	6	7	8	9
·00	1000	1002	1005	1007	1009	1012	1014	1016	1019	1021	2	0	0	1	1	1	1	1	2	2
·01	1023	1026	1028	1030	1033	1035	1038	1040	1042	1045		0	0	1	1	1	1	2	2	2
·02	1047	1050	1052	1054	1057	1059	1062	1064	1067	1069		0	0	1	1	1	1	2	2	2
·03	1072	1074	1076	1079	1081	1084	1086	1089	1091	1094		0	0	1	1	1	1	2	2	2
·04	1096	1099	1102	1104	1107	1109	1112	1114	1117	1119		0	1	1	1	1	2	2	2	2
·05	1122	1125	1127	1130	1132	1135	1138	1140	1143	1146		0	1	1	1	1	2	2	2	2
·06	1148	1151	1153	1156	1159	1161	1164	1167	1169	1172		0	1	1	1	1	2	2	2	2
·07	1175	1178	1180	1183	1186	1189	1191	1194	1197	1199		0	1	1	1	1	2	2	2	2
·08	1202	1205	1208	1211	1213	1216	1219	1222	1225	1227		0	1	1	1	1	2	2	2	3
·09	1230	1233	1236	1239	1242	1245	1247	1250	1253	1256		0	1	1	1	1	2	2	2	3
·10	1259	1262	1265	1268	1271	1274	1276	1279	1282	1285		0	1	1	1	1	2	2	2	3
·11	1288	1291	1294	1297	1300	1303	1306	1309	1312	1315	3	0	1	1	1	2	2	2	2	3
·12	1318	1321	1324	1327	1330	1334	1337	1340	1343	1346		0	1	1	1	2	2	2	2	3
·13	1349	1352	1355	1358	1361	1365	1368	1371	1374	1377		0	1	1	1	2	2	2	3	3
·14	1380	1384	1387	1390	1393	1396	1400	1403	1406	1409		0	1	1	1	2	2	2	3	3
·15	1413	1416	1419	1422	1426	1429	1432	1435	1439	1442		0	1	1	1	2	2	2	3	3
·16	1445	1449	1452	1455	1459	1462	1466	1469	1472	1476		0	1	1	1	2	2	2	3	3
·17	1479	1483	1486	1489	1493	1496	1500	1503	1507	1510		0	1	1	1	2	2	2	3	3
·18	1514	1517	1521	1524	1528	1531	1535	1538	1542	1545		0	1	1	1	2	2	2	3	3
·19	1549	1552	1556	1560	1563	1567	1570	1574	1578	1581		0	1	1	1	2	2	3	3	3
·20	1585	1589	1592	1596	1600	1603	1607	1611	1614	1618		0	1	1	1	2	2	3	3	3
·21	1622	1626	1629	1633	1637	1641	1644	1648	1652	1656		0	1	1	2	2	2	3	3	3
·22	1660	1663	1667	1671	1675	1679	1683	1687	1690	1694		0	1	1	2	2	2	3	3	3
·23	1698	1702	1706	1710	1714	1718	1722	1726	1730	1734	4	0	1	1	2	2	2	3	3	4
·24	1738	1742	1746	1750	1754	1758	1762	1766	1770	1774		0	1	1	2	2	2	3	3	4
·25	1778	1782	1786	1791	1795	1799	1803	1807	1811	1816		0	1	1	2	2	2	3	3	4
·26	1820	1824	1828	1832	1837	1841	1845	1849	1854	1858		0	1	1	2	2	3	3	3	4
·27	1862	1866	1871	1875	1879	1884	1888	1892	1897	1901		0	1	1	2	2	3	3	3	4
·28	1905	1910	1914	1919	1923	1928	1932	1936	1941	1945		0	1	1	2	2	3	3	4	4
·29	1950	1954	1959	1963	1968	1972	1977	1982	1986	1991		0	1	1	2	2	3	3	4	4
·30	1995	2000	2004	2009	2014	2018	2023	2028	2032	2037		0	1	1	2	2	3	3	4	4
·31	2042	2046	2051	2056	2061	2065	2070	2075	2080	2084		0	1	1	2	2	3	3	4	4
·32	2089	2094	2099	2104	2109	2113	2118	2123	2128	2133		0	1	1	2	2	3	3	4	4
·33	2138	2143	2148	2153	2158	2163	2168	2173	2178	2183	5	1	1	2	2	3	3	4	4	5
·34	2188	2193	2198	2203	2208	2213	2218	2223	2228	2234		1	1	2	2	3	3	4	4	5
·35	2239	2244	2249	2254	2259	2265	2270	2275	2280	2286		1	1	2	2	3	3	4	4	5
·36	2291	2296	2301	2307	2312	2317	2323	2328	2333	2339		1	1	2	2	3	3	4	4	5
·37	2344	2350	2355	2360	2366	2371	2377	2382	2388	2393		1	1	2	2	3	3	4	4	5
·38	2399	2404	2410	2415	2421	2427	2432	2438	2443	2449		1	1	2	2	3	3	4	4	5
·39	2455	2460	2466	2472	2477	2483	2489	2495	2500	2506		1	1	2	2	3	3	4	5	5
·40	2512	2518	2523	2529	2535	2541	2547	2553	2559	2564		1	1	2	2	3	4	4	5	5
·41	2570	2576	2582	2588	2594	2600	2606	2612	2618	2624	6	1	1	2	2	3	4	4	5	5
·42	2630	2636	2642	2649	2655	2661	2667	2673	2679	2685		1	1	2	2	3	4	4	5	6
·43	2692	2698	2704	2710	2716	2723	2729	2735	2742	2748		1	1	2	3	3	4	4	5	6
·44	2754	2761	2767	2773	2780	2786	2793	2799	2805	2812		1	1	2	3	3	4	4	5	6
·45	2818	2825	2831	2838	2844	2851	2858	2864	2871	2877		1	1	2	3	3	4	5	5	6
·46	2884	2891	2897	2904	2911	2917	2924	2931	2938	2944		1	1	2	3	3	4	5	5	6
·47	2951	2958	2965	2972	2979	2985	2992	2999	3006	3013		1	1	2	3	3	4	5	5	6
·48	3020	3027	3034	3041	3048	3055	3062	3069	3076	3083	7	1	1	2	3	4	4	5	6	6
·49	3090	3097	3105	3112	3119	3126	3133	3141	3148	3155		1	1	2	3	4	4	5	6	6

x	0	1	2	3	4	5	6	7	8	9	Δ	ADD								
												1	2	3	4	5	6	7	8	9
·50	3162	3170	3177	3184	3192	3199	3206	3214	3221	3228		1	1	2	3	4	4	5	6	7
·51	3236	3243	3251	3258	3266	3273	3281	3289	3296	3304		1	2	2	3	4	5	5	6	7
·52	3311	3319	3327	3334	3342	3350	3357	3365	3373	3381		1	2	2	3	4	5	5	6	7
·53	3388	3396	3404	3412	3420	3428	3436	3443	3451	3459		1	2	2	3	4	5	6	6	7
·54	3467	3475	3483	3491	3499	3508	3516	3524	3532	3540	8	1	2	2	3	4	5	6	6	7
·55	3548	3556	3565	3573	3581	3589	3597	3606	3614	3622		1	2	2	3	4	5	6	7	7
·56	3631	3639	3648	3656	3664	3673	3681	3690	3698	3707		1	2	3	3	4	5	6	7	8
·57	3715	3724	3733	3741	3750	3758	3767	3776	3784	3793		1	2	3	3	4	5	6	7	8
·58	3802	3811	3819	3828	3837	3846	3855	3864	3873	3882		1	2	3	4	4	5	6	7	8
·59	3890	3899	3908	3917	3926	3936	3945	3954	3963	3972	9	1	2	3	4	5	5	6	7	8
·60	3981	3990	3999	4009	4018	4027	4036	4046	4055	4064		1	2	3	4	5	6	6	7	8
·61	4074	4083	4093	4102	4111	4121	4130	4140	4150	4159		1	2	3	4	5	6	7	8	9
·62	4169	4178	4188	4198	4207	4217	4227	4236	4246	4256		1	2	3	4	5	6	7	8	9
·63	4266	4276	4285	4295	4305	4315	4325	4335	4345	4355	10	1	2	3	4	5	6	7	8	9
·64	4365	4375	4385	4395	4406	4416	4426	4436	4446	4457		1	2	3	4	5	6	7	8	9
·65	4467	4477	4487	4498	4508	4519	4529	4539	4550	4560		1	2	3	4	5	6	7	8	9
·66	4571	4581	4592	4603	4613	4624	4634	4645	4656	4667		1	2	3	4	5	6	7	9	10
·67	4677	4688	4699	4710	4721	4732	4742	4753	4764	4775	11	1	2	3	4	6	7	8	9	10
·68	4786	4797	4808	4819	4831	4842	4853	4864	4875	4887		1	2	3	4	6	7	8	9	10
·69	4898	4909	4920	4932	4943	4955	4966	4977	4989	5000		1	2	3	5	6	7	8	9	10
·70	5012	5023	5035	5047	5058	5070	5082	5093	5105	5117		1	2	4	5	6	7	8	9	11
·71	5129	5140	5152	5164	5176	5188	5200	5212	5224	5236	12	1	2	4	5	6	7	8	10	11
·72	5248	5260	5272	5284	5297	5309	5321	5333	5346	5358		1	2	4	5	6	7	9	10	11
·73	5370	5383	5395	5408	5420	5433	5445	5458	5470	5483		1	3	4	5	6	8	9	10	11
·74	5495	5508	5521	5534	5546	5559	5572	5585	5598	5610		1	3	4	5	6	8	9	10	12
·75	5623	5636	5649	5662	5675	5689	5702	5715	5728	5741	13	1	3	4	5	7	8	9	10	12
·76	5754	5768	5781	5794	5808	5821	5834	5848	5861	5875		1	3	4	5	7	8	9	11	12
·77	5888	5902	5916	5929	5943	5957	5970	5984	5998	6012		1	3	4	5	7	8	10	11	12
·78	6026	6039	6053	6067	6081	6095	6109	6124	6138	6152	14	1	3	4	6	7	8	10	11	13
·79	6166	6180	6194	6209	6223	6237	6252	6266	6281	6295		1	3	4	6	7	9	10	11	13
·80	6310	6324	6339	6353	6368	6383	6397	6412	6427	6442		1	3	4	6	7	9	10	12	13
·81	6457	6471	6486	6501	6516	6531	6546	6561	6577	6592	15	2	3	5	6	8	9	11	12	14
·82	6607	6622	6637	6653	6668	6683	6699	6714	6730	6745		2	3	5	6	8	9	11	12	14
·83	6761	6776	6792	6808	6823	6839	6855	6871	6887	6902		2	3	5	6	8	9	11	13	14
·84	6918	6934	6950	6966	6982	6998	7015	7031	7047	7063	16	2	3	5	6	8	10	11	13	14
·85	7079	7096	7112	7129	7145	7161	7178	7194	7211	7228		2	3	5	7	8	10	12	13	15
·86	7244	7261	7278	7295	7311	7328	7345	7362	7379	7396	17	2	3	5	7	9	10	12	14	15
·87	7413	7430	7447	7464	7482	7499	7516	7534	7551	7568		2	3	5	7	9	10	12	14	16
·88	7586	7603	7621	7638	7656	7674	7691	7709	7727	7745		2	4	5	7	9	11	12	14	16
·89	7762	7780	7798	7816	7834	7852	7870	7889	7907	7925	18	2	4	5	7	9	11	13	14	16
·90	7943	7962	7980	7998	8017	8035	8054	8072	8091	8110		2	4	6	7	9	11	13	15	17
·91	8128	8147	8166	8185	8204	8222	8241	8260	8279	8299	19	2	4	6	8	10	11	13	15	17
·92	8318	8337	8356	8375	8395	8414	8433	8453	8472	8492		2	4	6	8	10	12	14	15	17
·93	8511	8531	8551	8570	8590	8610	8630	8650	8670	8690		2	4	6	8	10	12	14	16	18
·94	8710	8730	8750	8770	8790	8810	8831	8851	8872	8892	20	2	4	6	8	10	12	14	16	18
·95	8913	8933	8954	8974	8995	9016	9036	9057	9078	9099		2	4	6	8	10	12	15	17	19
·96	9120	9141	9162	9183	9204	9226	9247	9268	9290	9311	21	2	4	6	8	11	13	15	17	19
·97	9333	9354	9376	9397	9419	9441	9462	9484	9506	9528		2	4	7	9	11	13	15	17	20
·98	9550	9572	9594	9616	9638	9661	9683	9705	9727	9750	22	2	4	7	9	11	13	15	18	20
·99	9772	9795	9817	9840	9863	9886	9908	9931	9954	9977	23	2	5	7	9	11	14	16	18	21

7.2 Slide Rules

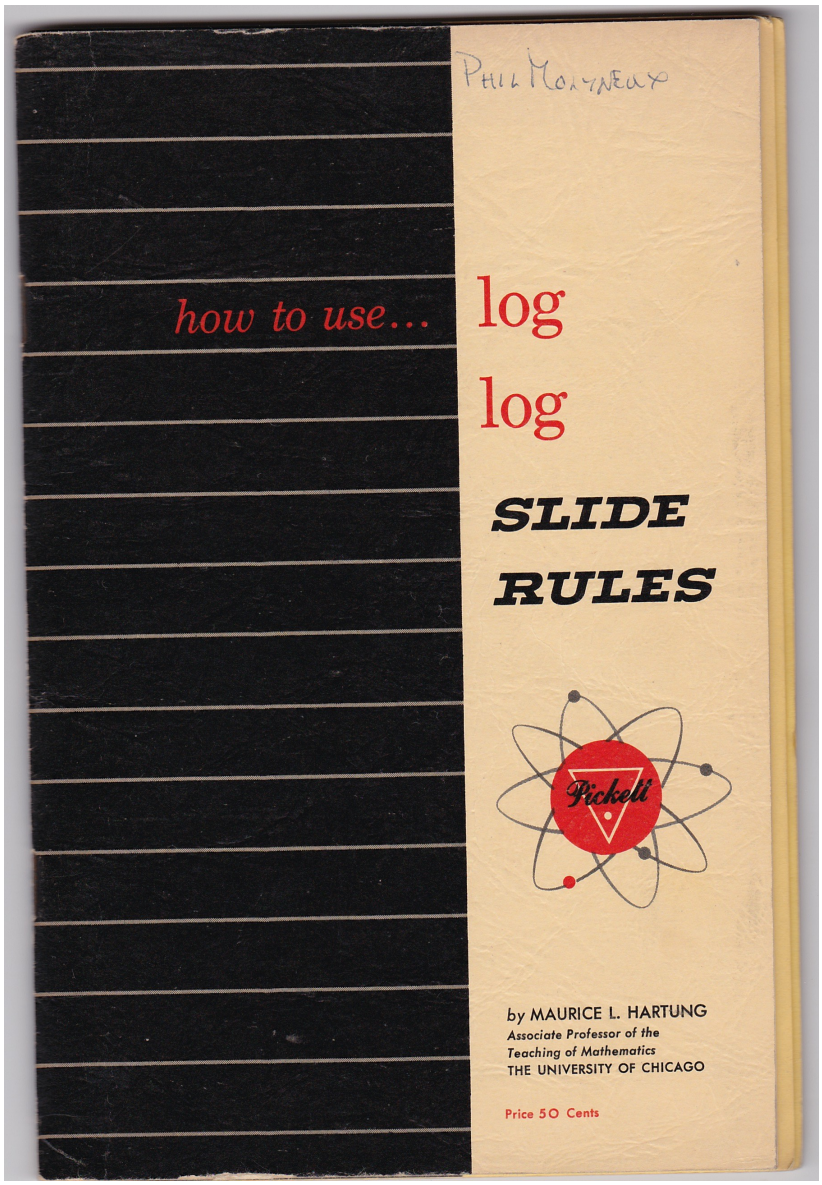
Pickett N 3-ES from 1967





- See Oughtred Society
- UKSRC
- Rod Lovett's Slide Rules
- Slide Rule Museum

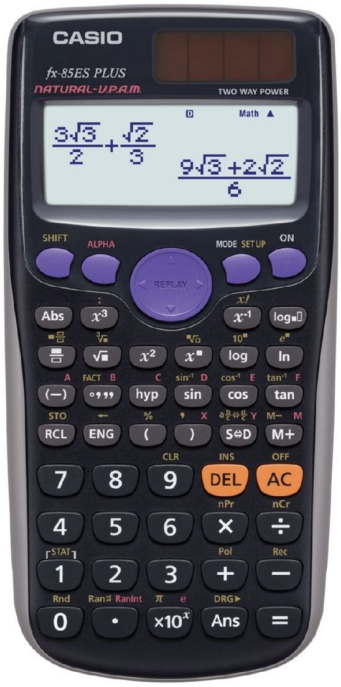
Pickett log log **Slide Rules Manual 1953**



7.3 Calculators

HP HP-21 Calculator from 1975 £69

Casio fx-85GT PLUS Calculator from 2013 £10



Calculator links

- **HP Calculator Museum** http://www.hpmuseum.org
- **HP Calculator Emulators** http://nonpareil.brouhaha.com
- **HP Calculator Emulators for OS X** http://www.bartosiak.org/nonpareil/
- **Vintage Calculators Web Museum** http://www.vintagecalculators.com

7.4 Example Calculation

- Evaluate 89.7×597
- **Knott's Tables**
- $\log_{10} 89.7 = 1.9528$ and $\log_{10} 597 = 2.7760$
- Shows *mantissa* (decimal) & *characteristic* (integral)
- Add 4.7288, take *antilog* to get $5346 + 10 = 5.356 \times 10^4$
- **HP-21 Calculator** — set display to 4 decimal places
- 89.7 [log] $= 1.9528$ and 597 [log] $= 2.7760$
- [+] displays 4.7288
- 10 [ENTER], [$x \rightleftarrows y$] and [y^x] displays 53550.9000
- **Casio fx-85GT PLUS**
- [log] 89.7 [)] $= 1.952792443$ [+] [log] 597 [)] $= 2.775974331$ [=]
- 4.728766774 [Ans] [+] [10^x] gives 53550.9

8 Logic and Truth Tables

8.1 Boolean Expressions and Truth Tables

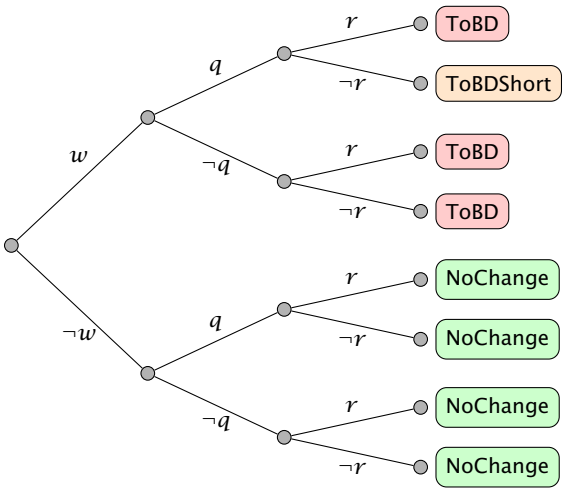
Traffic Lights Example

- Consider traffic light at the intersection of roads AC and BD with the following rules for the AC controller
- Vehicles should not wait on red on BD for too long.
- If there is a long queue on AC then BD is only given a green for a short interval.
- If both queues are long the usual flow times are used.
- We use the following propositions:
  - w Vehicles have been waiting on red on BD for too long
  - q Queue on AC is too long
  - r Queue on BD is too long

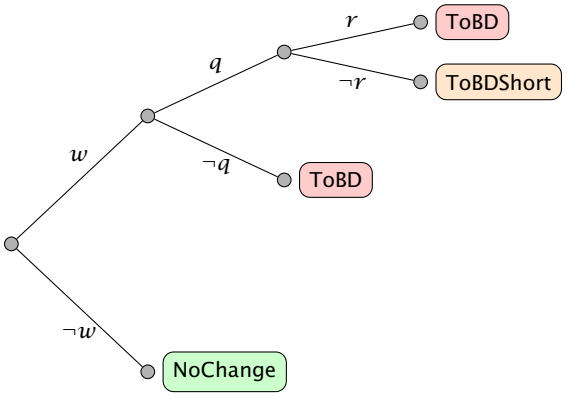
- Given the following events:
  - ToBD Change flow to BD
  - ToBDShort Change flow to BD for short time
  - NoChange No Change to lights
- Express above as truth table, outcome tree, boolean expression
- **Traffic Lights outcome table**

w	q	r	Event
T	T	T	ToBD
T	T	F	ToBDShort
T	F	T	ToBD
T	F	F	ToBD
F	T	T	NoChange
F	T	F	NoChange
F	F	T	NoChange
F	F	F	NoChange

- **Traffic lights outcome tree**



- **Traffic lights outcome tree** simplified



- **Traffic Lights code 01**
- See M269TutorialProgPythonADT01.py

```
3 def trafficLights01(w,q,r) :
4   """
```

```
  Input 3 Booleans
  Return Event string
  """
  if w :
    if q :
      if r :
        evnt = "ToBD"
      else :
        evnt = "ToBDShort"
    else :
      evnt = "ToBD"
  else :
    evnt = "NoChange"
  return evnt
```

- **Traffic Lights test code 01**

```
trafficLights01Evnts = [((w,q,r), trafficLights01(w,q,r))
                            for w in [True,False]
                            for q in [True,False]
                            for r in [True,False]]

assert trafficLights01Evnts \
  == [((True, True, True), 'ToBD')
       ,((True, True, False), 'ToBDShort')
       ,((True, False, True), 'ToBD')
       ,((True, False, False), 'ToBD')
       ,((False, True, True), 'NoChange')
       ,((False, True, False), 'NoChange')
       ,((False, False, True), 'NoChange')
       ,((False, False, False), 'NoChange')]
```

- **Traffic Lights code 02** compound Boolean conditions

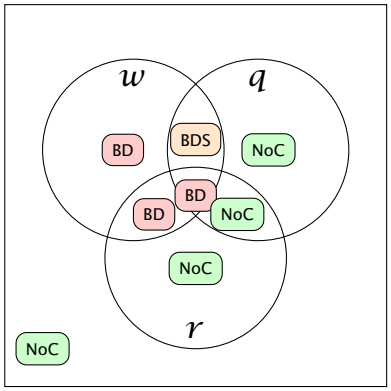
```
def trafficLights02(w,q,r) :
  """
  Input 3 Booleans
  Return Event string
  """
  if ((w and q and r) or (w and not q)) :
    evnt = "ToBD"
  elif (w and q and not r) :
    evnt = "ToBDShort"
  else :
    evnt = "NoChange"
  return evnt
```

- What objectives do we have for our code ?

- **Traffic Lights test code 02**

```
trafficLights02Evnts = [((w,q,r), trafficLights02(w,q,r))
                            for w in [True,False]
                            for q in [True,False]
                            for r in [True,False]]

assert trafficLights02Evnts == trafficLights01Evnts
```



- **Traffic Lights Venn diagram**
- OK using a fill colour would look better but didn't have the time to hack the package

8.2 Conditional Expressions and Validity

- **Validity** of Boolean expressions
- **Complete** every outcome returns an event (or error message, raises an exception)
- **Consistent** — we do not want two nested `if` statements or expressions resulting in different events
- We check this by ensuring that the events form a disjoint partition of the set of outcomes — see the Venn diagram
- We would quite like the programming language processor to warn us otherwise — not always possible

8.3 Boolean Expressions Exercise

Rail Ticket Exercise

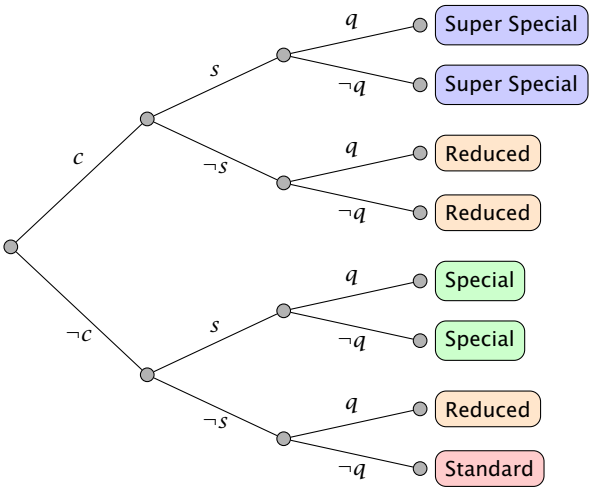
- Rail ticket discounts for:
  - c Rail card
  - q Off-peak time
  - s Special offer
- 4 fares: Standard, Reduced, Special, Super Special
- Rules:
  1. Reduced fare if rail card or at off-peak time
  2. Without rail card no reduction for both special offer and off-peak.
  3. Rail card always has reduced fare but cannot get off-peak discount as well.
  4. Rail card gets super special discount for journey with special offer
- Draw up truth table, outcome tree, Venn diagram and conditional statement (or expression) for this
- Rail ticket outcome table

c	q	s	Event
T	T	T	Super Special
T	T	F	Reduced
T	F	T	Super Special
T	F	F	Reduced
F	T	T	Special
F	T	F	Reduced
F	F	T	Special
F	F	F	Standard

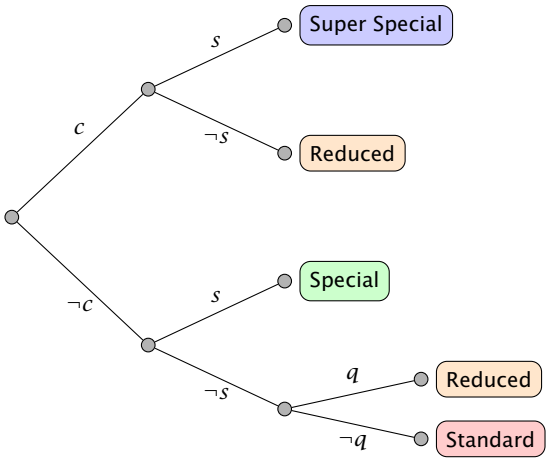
- Rail ticket outcome table
- Note that it may be more convenient to change columns

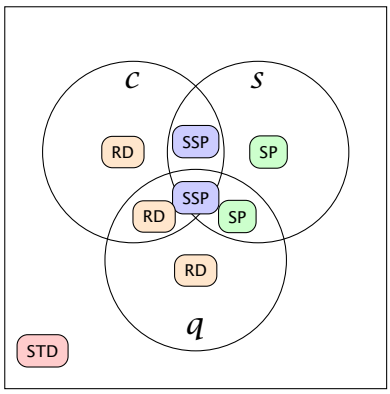
c	s	q	Event
T	T	T	Super Special
T	T	F	Super Special
T	F	T	Reduced
T	F	F	Reduced
F	T	T	Special
F	T	F	Special
F	F	T	Reduced
F	F	F	Standard

- Real fares are a little more complex — see brfares.com
- **Rail Ticket outcome tree**



- **Rail Ticket outcome tree** simplified





- **Rail Ticket Venn diagram**
- **Rail Ticket code 01**

```
61 def railTicket01(c,s,q) :
62   """
63   Input 3 Booleans
64   Return Event string
65   """
66   if c :
67     if s :
68       evnt = "SSP"
69     else :
70       evnt = "RD"
71   else :
72     if s :
73       evnt = "SP"
74     else :
75       if q :
76         evnt = "RD"
77       else :
78         evnt = "STD"
79   return evnt
```

- **Rail Ticket test code 01**

```
83 railTicket01Evnts = [((c,s,q), railTicket01(c,s,q))
84                               for c in [True,False]
85                               for s in [True,False]
86                               for q in [True,False]]
87
88 assert railTicket01Evnts \
89   == [((True, True, True), 'SSP')
90       ,((True, True, False), 'SSP')
91       ,((True, False, True), 'RD')
92       ,((True, False, False), 'RD')
93       ,((False, True, True), 'SP')
94       ,((False, True, False), 'SP')
95       ,((False, False, True), 'RD')
96       ,((False, False, False), 'STD')]
```

- **Rail Ticket code 02** compound Boolean expressions

```
98 def railTicket02(c,s,q) :
99   """
100  Input 3 Booleans
101  Return Event string
102  """
103  if (c and s) :
104    evnt = "SSP"
105  elif ((c and not s) or (not c and not s and q)) :
106    evnt = "RD"
107  elif (not c and s) :
108    evnt = "SP"
109  else :
110    evnt = "STD"
111  return evnt
```

- **Rail Ticket test code 02**

```
railTicket02Evnts = [((c,s,q), railTicket02(c,s,q))
                             for c in [True,False]
                             for s in [True,False]
                             for q in [True,False]]

assert railTicket02Evnts == railTicket01Evnts
```

8.4 Propositional Calculus

- Unit 2 section 3.2 *A taste of formal logic* introduces Propositional calculus
- A language for calculating about Booleans — *truth values*
- Gives operators (connectives) conjunction ($\wedge$) AND, disjunction ($\vee$) OR, negation ($\neg$) NOT, implication ($\Rightarrow$) IF
- There are 16 possible functions $(\mathbb{B}, \mathbb{B}) \to \mathbb{B}$ — see below — defined by their truth tables
- **Discussion** Did you find the truth table for implication weird or surprising ?
- **Implication** has a **negative** definition — we accept its truth unless we have contrary evidence
- $T \Rightarrow T == T$ and $T \Rightarrow F == F$
- Hence 4 possibilities for truth table

p	q	$p \Uparrow q$	q	$p \Updownarrow q$	$p < q$
T	T	T	T	T	T
T	F	F	F	F	F
F	T	T	T	F	F
F	F	T	F	T	F

- ($\Rightarrow$) must have the entry shown — the others are taken
- Do not think of p *causing* q
- **Functionally complete** set of connectives is one which can be used to express all possible connectives
- $p \Rightarrow q \equiv \neg p \vee q$ so we could just use $\{\neg, \wedge, \vee\}$
- **Boolean programming** — we have to have a functionally complete set but choose more to make the programming easier
- *Expressiveness* is an issue in programming language design
- **NAND** $p \overline{\wedge} q$, $p \uparrow q$, Sheffer stroke
- **NOR** $p \overline{\vee} q$, $p \downarrow q$, Pierce's arrow
- See truth tables below — both $\{\uparrow\}, \{\downarrow\}$ are functionally complete
- **Exercise** verify

  - $\neg p \equiv p \uparrow p$
  - $p \wedge q \equiv \neg(p \uparrow q) = (p \uparrow q) \uparrow (p \uparrow q)$
  - $p \vee q \equiv (p \uparrow p) \uparrow (q \uparrow q)$
  - $\neg p \equiv p \downarrow p$
  - $p \wedge q \equiv (p \downarrow p) \downarrow (q \downarrow q)$
  - $p \vee q \equiv \neg(p \downarrow q) = (p \downarrow q) \downarrow (p \downarrow q)$
- Not a novelty — the Apollo Guidance Computer was implemented in NOR gates alone.

8.5 Truth Function

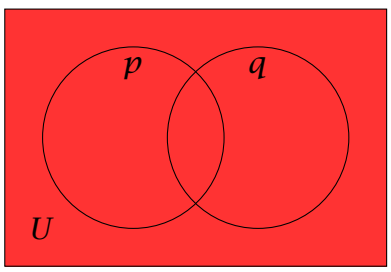
- The following appendix notes illustrate the 16 binary functions of two Boolean variables
- See Truth function
- See Functional completeness
- See Sheffer stroke
- See Logical NOR

Table of Binary Truth Functions

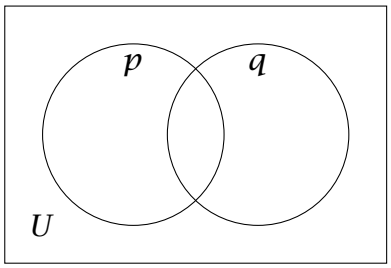
p	q	$\top$	$p \veebar q$	$p \nLeftarrow q$	p	$p \Rightarrow q$	q	$p \Leftrightarrow q$	$p \barwedge q$
T	T	T	T	T	T	T	T	T	T
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

p	q	$\bot$	$p \barvee q$	$p \Leftarrow q$	$\neg p$	$p \nRightarrow q$	$\neg q$	$p \nLeftrightarrow q$	$p \veebar q$
T	T	F	F	F	F	F	F	F	F
T	F	F	F	F	F	T	T	T	T
F	T	F	F	T	T	F	F	T	T
F	F	F	T	F	T	F	T	F	T

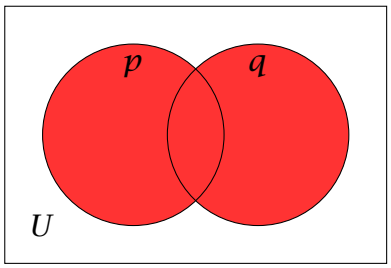
- **Tautology** True, $\top$, *Top*



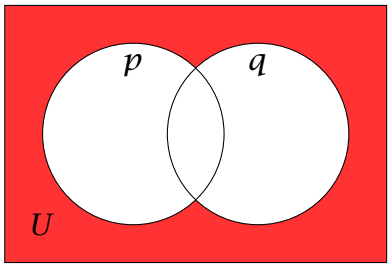
- **Contradiction** False, $\bot$, *Bottom*



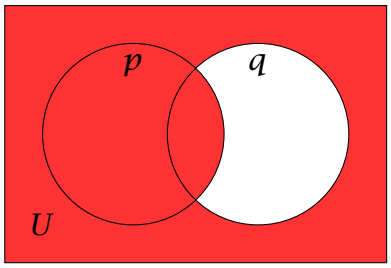
- **Disjunction** OR, $p \vee q$



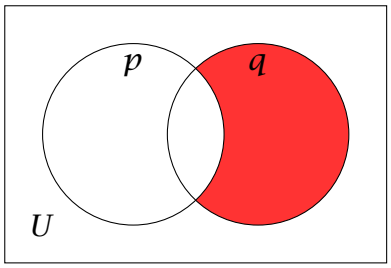
- **Joint Denial** NOR, $p \overline{\vee} q$, $p \downarrow q$, *Pierce's arrow*



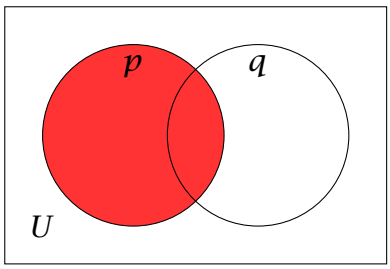
- **Converse Implication** $p \Leftarrow q$



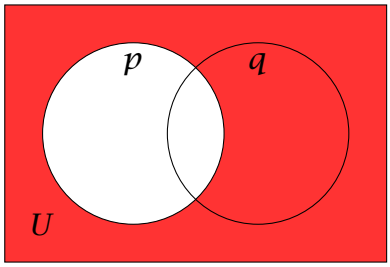
- **Converse Nonimplication** $p \not\Leftarrow q$



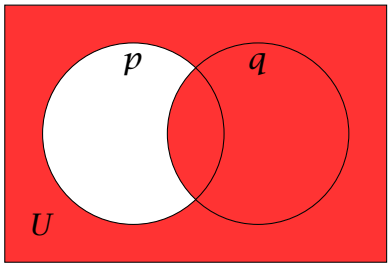
- **Proposition** p



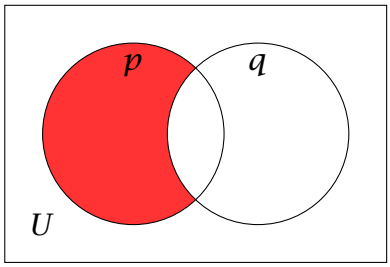
- **Negation of** p



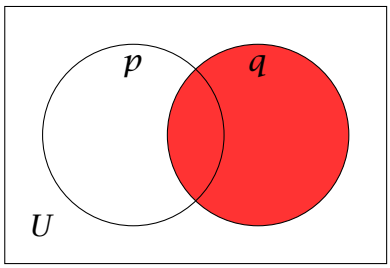
- **Material Implication** $p \Rightarrow q$



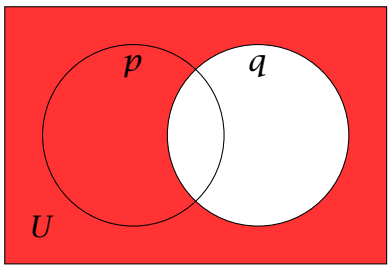
- **Material Nonimplication** $p \nRightarrow q$



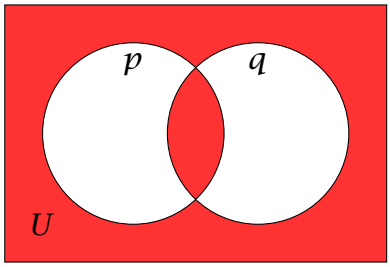
- **Proposition** q q



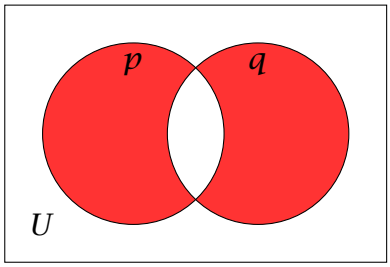
- **Negation of** q $\neg q$



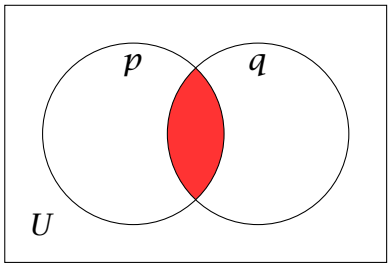
- **Biconditional** If and only if, IFF, $p \Leftrightarrow q$



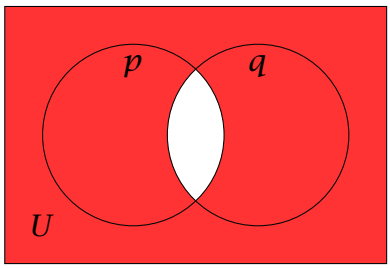
- **Exclusive disjunction** XOR, $p \nLeftrightarrow q$



- **Conjunction** AND, $p \wedge q$



- **Alternative denial** NAND, $p \barwedge q$, $p \uparrow q$, *Sheffer stroke*



9 Abstract Data Types

9.1 Abstract Data Types — Overview

- **Abstract data type** is a type with associated operations, but whose representation is hidden (or not accessible)
- Common examples in most programming languages are Integer and Characters and other built in types such as tuples and lists
- Abstract data types are modeled on Algebraic structures
  - A set of values
  - Collection of operations on the values
  - Axioms for the operations may be specified as equations or pre and post conditions
- **Health Warning** different languages provide different ways of doing data abstraction with similar names that may mean subtly different things
- **Abstract Data Types and Object-Oriented Programming**
- Example: Shape with `Circle`s, `Square`s, ... and operations `draw`, `moveTo`, ...
- **ADT** approach centres on the data type — that tells you what shapes exist
- For each operation on shapes, you describe what they do for different shapes.
- **OO** you declare that to be a shape, you have to have some operations (`draw`, `moveTo`)
- For each kind of shape you provide an implementation of the operations
- **OO** easier to answer *What is a circle?* and add new shapes
- **ADT** easier to answer *How do you draw a shape?* and add new operations
- **Health Warning and Optional Material** Discussions about the merits of Functional programming and Object-oriented programming tend to look like the disputes between Lilliput and Blefuscu
- Abstract data type article contrasts ADT and OO as algebra compared to co-algebra
- What does *coalgebra* mean in the context of programming? is a fairly technical but accessible article.
- What does the *forall* keyword in Haskell do? — is an accessible article on *Existential Quantification*
- Bart Jacobs Coalgebra
- nLab Coalgebra
- Beware the distinction between *concepts* and *features* in programming languages — see OOP Disaster
- **Not for this session** — this slide is here just in case

Further Links on ADT/OOP

- Object Oriented Programming in Haskell (15 October 2018)
- Object-Oriented Haskell (15 October 2018)

```haskell
data Shape
  = Circle Point Radius
   | Square Point Size

draw :: Shape -> Pict
draw (Circle p r) = drawCircle p r
draw (Square p s) = drawRectangle p s s

moveTo :: Point -> Shape -> Shape
moveTo p2 (Circle p1 r) = Circle p2 r
moveTo p2 (Square p1 s) = Square p2 s

shapes :: [Shape]
shapes = [Circle (0,0) 1, Square (1,1) 2]

shapes01 :: [Shape]
shapes01 = map (moveTo (2,2)) shapes
```

- Example based on Lennart Augustsson email of 23 June 2005 on Haskell list

```haskell
class IsShape shape where
  draw :: shape -> Pict
  moveTo :: Point -> shape -> shape

data Shape = forall a . (IsShape a) => Shape a

data Circle = Circle Point Radius
instance IsShape Circle where
  draw (Circle p r) = drawCircle p r
  moveTo p2 (Circle p1 r) = Circle p2 r

data Square = Square Point Size
instance IsShape Square where
  draw (Square p s) = drawRectangle p s s
  moveTo p2 (Square p1 s) = Square p2 s

shapes :: [Shape]
shapes = [Shape (Circle (0,0) 10), Shape (Square (1,1) 2)]

shapes01 :: [Shape]
shapes01 = map (moveShapeTo (2,2)) shapes
           where
           moveShapeTo p (Shape s) = Shape (moveTo p s)
```

The Expression Problem

- The *Expression Problem* describes a dual problem that neither Object Oriented Programming nor Functional Programming fully addresses.
- If you want to add a new thing, Object Oriented Programming makes it easy (since you can simply create a new class) but Functional Programming makes it harder (since you have to edit every function that accepts a thing of that type)
- If you want to add a new function, Functional Programming makes it easy (simply add a new function) while Object Oriented Programming makes it harder (since you have to edit every class to add the function)
- Wikipedia: Expression problem
- Bendersky: The Expression Problem and More thoughts
- C2 Wiki: Expression Problem

- What is the 'expression problem'?
- Philip Wadler: The Expression Problem

9.2 Abstract Data Type — Queue

- Queue Abstract Data Type — operations
- `makeEmptyQ` returns empty queue
- `isEmptyQ` takes queue, returns Boolean
- `addToQ` takes queue, item, returns queue with item added at back
- `headOfQ` takes queue, returns item at front
- `tailOfQ` takes queue, returns queue without front item
- Other operations
- `removeFrontQ` takes queue, returns pair of item on the front and queue with item removed
- `sizeQ` to save calculating it
- `isFullQ` for a bounded queue
- **Pre, Post Conditions, Axioms** should be **complete**
- They define all permissable inputs to the functions (or methods)
- They define the outcome of all applications of the functions
- Composition of the functions constructs all possible members of the ADT set
- **Pre-conditions, Post-conditions, Axioms**
- `makeEmptyQ()`
- **Pre** `True`
- **Post** Return value `q` is an empty queue
- **Axiom** `makeEmptyQ() == EmptyQ`
- `isEmptyQ()`
- **Pre** `True`
- **Post** Returns `True` if `q` is empty, otherwise `False`
- **Axiom** `isEmptyQ(makeEmptyQ()) == True`
- `isEmptyQ(addToQ(q,x)) == False`
- **Exercise** complete this for the other operations
- **Pre-conditions, Post-conditions, Axioms**
- `addToQ()`
- **Pre** `True`
- **Post** Returns queue with `x` at back, front part is input queue

- `headOfQ()`
- **Pre** Argument `q` is non-empty
- **Post** Return value is item at the front (queue is unchanged)
- **Axioms** `headOfQ(makeEmptyQ()) == error`
- `headOfQ(addToQ(makeEmptyQ(),x)) == x`
- `headOfQ(addToQ(q,x)) == headOfQ(q)`
- **Pre-conditions, Post-conditions, Axioms**
- `tailOfQ()`
- **Pre** `True`
- **Post** Returns queue without first item
- **Axioms** `tailOfQ(makeEmptyQ()) == error`
- `tailOfQ(addToQ(makeEmptyQ(),x)) == EmptyQ`
- `tailOfQ(addToQ(q,x)) == addToQ(tailOfQ(q),x)`
- **Queue Implementation**
- Using Lists as Queues section 5.1.2 of the *Tutorial*
- Quote: It is also possible to use a list as a queue, where the first element added is the first element retrieved (first-in, first-out); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).
- Could use `collections.deque` but we will use a pair of lists — See (Okasaki, 1998, page 42)
- **Queue Implementation 1**
- Using a `namedtuple()`
- A factory function for creating tuple subclasses with named fields

```
5  from collections import namedtuple

7  Qp1 = namedtuple('Qp1',['frs','rbks'])
```

- **Queue Implementation 1** main operations

```
9  def makeEmptyQp1():
10   return Qp1([],[])

12 def isEmptyQp1(q):
13   return q.frs == []

15 def addToQp1(q,x):
16   return checkQp1(q.frs, [x] + q.rbks[:])

18 def headOfQp1(q):
19   if q.frs == [] :
20     RunTimeError("headOfQp1 applied to empty queue")
21   else:
22     return q.frs[0]

24 def tailOfQp1(q):
25   if q.frs == [] :
26     RunTimeError("tailOfQp1 applied to empty queue")
27   else:
28     return checkQp1(q.frs[1:], q.rbks[:])
```

- **Queue Implementation 1** checkOp1()

```
30 def checkQp1(frs, rbks):
31   if frs == [] :
32     bks = rbks[:]
33     bks.reverse()
34     return Qp1(bks, [])
35   else :
36     return Qp1(frs, rbks)
```

- Note copying of arguments — see below for reason
- Note also in stringQp1Items below at line 47 on page 45
- implicit line joining using (()) (why is this needed ??)
- Note use of recursion

Python Argument Passing

- **Functions, Immutable and Mutable Arguments**
- Immutable arguments are passed by value
- Mutable arguments are passed by reference
- Immutable: numbers, strings, tuples
- Mutable: Lists, dictionaries, sets, and most objects in user classes

```
>>> def changer (a,b) :
...   a = 2
...   b[0] = 'spam'
...
>>> n = 1
>>> xs = [1,2]
>>> changer(n, xs)
>>> (n,xs)
(1, ['spam', 2])
```

- **Queue Implementation 1** conversion operations

```
38 def stringQp1(q) :
39   return ("<" + stringQp1Items(q) + ">")

41 def stringQp1Items(q) :
42   if isEmptyQp1(q) :
43     return ""
44   elif isEmptyQp1(tailOfQp1(q)) :
45     return str(headOfQp1(q))
46   else :
47     return ( str(headOfQp1(q))
48              + ",␣" + stringQp1Items(tailOfQp1(q)) )

50 def buildQp1(xs,q) :
51   if xs == [] :
52     return q
53   else :
54     return buildQp1(xs[1:],addToQp1(q,xs[0]))

56 def listToQp1(xs) :
57   return buildQp1(xs, makeEmptyQp1())
```

- **Queue Implementation 1** test code

```
61 q11 = listToQp1([1,2,3,1])

63 q12 = tailOfQp1(q11)
```

```
65 assert q11 == Qp1(frs=[1], rbks=[1, 3, 2])

67 assert stringQp1(q11) == '<1,␣2,␣3,␣1>'

69 assert q12 == Qp1(frs=[2, 3, 1], rbks=[])

71 assert stringQp1(q12) == '<2,␣3,␣1>'
```

- **Queue Implementation 2**
- Modify to add `size`
- Store in tuple to save calculating each time

```
75 Qp2 = namedtuple('Qp2',['frs','rbks','sz'])
```

- **Exercise** Add `size()` operation and other modifications
- **Queue Implementation 2** main operations

```
77 def makeEmptyQp2():
78   return Qp2([],[], 0)

80 def isEmptyQp2(q):
81   return q.frs == []

83 def addToQp2(q,x):
84   return checkQp2(q.frs, [x] + q.rbks[:], q.sz + 1)

86 def headOfQp2(q):
87   if q.frs == [] :
88     RunTimeError("headOfQp2␣applied␣to␣empty␣queue")
89   else:
90     return q.frs[0]

92 def tailOfQp2(q):
93   if q.frs == [] :
94     RunTimeError("tailOfQp2␣applied␣to␣empty␣queue")
95   else:
96     return checkQp2(q.frs[1:], q.rbks[:], q.sz - 1)
```

- **Queue Implementation 2** `sizeQp2()`, `checkOp1()`

```
98 def sizeOfQp2(q) :
99   return q.sz

101 def checkQp2(frs, rbks, sz):
102   if frs == [] :
103     bks = rbks[:]
104     bks.reverse()
105     return Qp2(bks, [], sz)
106   else :
107     return Qp2(frs, rbks, sz)
```

- Note also in `stringQp2Items` below at line 118 on page 47
- implicit line joining using (()) (why is this needed ??)
- Note use of recursion
- **Queue Implementation 2** conversion operations

```
109 def stringQp2(q) :
110   return ("<" + stringQp2Items(q) + ">")

112 def stringQp2Items(q) :
113   if isEmptyQp2(q) :
114     return ""
115   elif isEmptyQp2(tailOfQp2(q)) :
116     return str(headOfQp2(q))
117   else :
```

```
118     return ( str(headOfQp2(q))
119             + ", " + stringQp2Items(tailOfQp2(q)) )

121 def buildQp2(xs,q) :
122   if xs == [] :
123     return q
124   else :
125     return buildQp2(xs[1:],addToQp2(q,xs[0]))

127 def listToQp2(xs) :
128   return buildQp2(xs, makeEmptyQp2())
```

- **Queue Implementation 2** test code

```
132 q21 = listToQp2([1,2,3,1])

134 q22 = tailOfQp2(q21)

136 assert q21 == Qp2(frs=[1], rbks=[1, 3, 2], sz=4)

138 assert stringQp2(q21) == '<1, 2, 3, 1>'

140 assert q22 == Qp2(frs=[2, 3, 1], rbks=[], sz=3)

142 assert stringQp2(q22) == '<2, 3, 1>'
```

9.3 ADT Lists in Lists

- Lists implemented naively as linked lists have some operations that take constant time and some that are linear in the length of the list
- Adding an element to the front of a list takes constant time while adding an element to the rear takes linear time
- This section reimplements lists using a pair of lists that overcomes this asymmetry in efficiency giving constant time for all operations.
- The basic idea is quite simple: break the list in two and reverse the second half
- This means that the last element is the first element of the second list
- A problem arises when one attempts to remove an element — in some cases the list has to be reorganised into two halves
- The criteria for reorganising gives the clue in how to write the code
- This implementation is based on Bird and Gibbons (2020, chp 3)
- The idea is attributed to Gries (1981, page 250) and Hood and Melville (1980)
- See also Hoogerwoord (1992)
- We give the code in Python from `SymmetricLists.py` with Haskell type specifications and declarations given as comments
- Here is the type alias declaration as a comment along with `fromSL` which converts back from symmetric lists to standard lists — this is known as the *abstraction function*

```
12 #  type SymList a = ([a],[a])

14 # Abstraction function

16 # fromSL :: SymList a -> [a]
```

```
18 def fromSL (pr) :
19   xs = pr[0]
20   ys = pr[1]
21   return xs + reverseF (ys)

23 def reverseF (xs) :
24   ys = xs[:]
25   ys.reverse()
26   return ys
```

- The abstraction function captures the relationship between the implementation of an operation on the representing type and its abstract type with an equation
- The *Eureka* bit of the implementation is spotting the *representation invariant* that our definitions both exploit and maintain

```
28 # repInvSL :: SymList a -> Bool

30 def repInvSL (pr) :
31   xs = pr[0]
32   ys = pr[1]
33   xsTest = ((not isEmpty (xs))
34             or (isEmpty (ys) or singleton (ys)))
35   ysTest = ((not isEmpty (ys))
36             or (isEmpty (xs) or singleton (xs)))
37   return (xsTest and ysTest)
```

- This says if one list is empty then the other must be either empty or a singleton
- This tells us when we need to reorganise the lists
- Here are the service operations for empty lists and singletons

```
39 # isEmpty :: [a] -> Bool

41 def isEmpty (xs) :
42   return (xs == [])

44 # isEmptySL :: SymList a -> Bool

46 def isEmptySL (pr) :
47   xs = pr[0]
48   ys = pr[1]
49   return (isEmpty (xs) and isEmpty (ys))

51 # singleton :: [a] -> Bool

53 def singleton (xs) :
54   return (len(xs) == 1)

56 # singletonSL :: SymList a -> Bool

58 def singletonSL (pr) :
59   xs = pr[0]
60   ys = pr[1]
61   return ((isEmpty (xs) and singleton (ys))
62           or (isEmpty (ys) and singleton (xs)))
```

- Constructor operations
- Both of these definitions make use of the representation invariant

```
64 #  Constructor functions

66 # consSL :: a -> SymList a -> SymList a

68 def consSL (x, pr) :
69   xs = pr[0]
70   ys = pr[1]
71   if isEmpty (ys) :
72     return ([x],xs)
```

```
  else :
    return ([x] + xs, ys)

# snocSL :: a -> SymList a -> SymList a

def snocSL (x, pr) :
  xs = pr[0]
  ys = pr[1]
  if isEmpty (xs) :
    return (ys,[x])
  else :
    return (xs, [x] + ys)
```

- Inspectors

```
# headSL :: SymList a -> a

def headSL (pr) :
  xs = pr[0]
  ys = pr[1]
  if isEmpty (xs) :
    if isEmpty (ys) :
      raise RuntimeError("headSL ([],[])")
    else :
      return ys[0]
  else :
    return xs[0]

# lastSL :: SymList a -> a

def lastSL (pr) :
  xs = pr[0]
  ys = pr[1]
  if isEmpty (ys) :
    if isEmpty (xs) :
      raise RuntimeError("tailSL ([],[])")
    else :
      return xs[0]
  else :
    return ys[0]
```

- tailSL
- Notice how the representation invariant is maintained

```
# tailSL :: SymList a -> SymList a

def tailSL (pr) :
  xs = pr[0]
  ys = pr[1]
  if isEmpty (xs) :
    if isEmpty (ys):
      raise RuntimeError("tailSL ([],[])")
    else:
      return ([],[])
  elif singleton (xs) :
    splitPt = len(ys) // 2
    (us,vs) = (ys[:splitPt],ys[splitPt:])
    return (reverseF (vs), us)
  else :
    return (xs[1:],ys)
```

- initSL

```
# initSL :: SymList a -> SymList a

def initSL (pr) :
  xs = pr[0]
  ys = pr[1]
  if isEmpty (ys) :
    if isEmpty (xs):
      raise RuntimeError("initSL ([],[])")
    else:
      return ([],[])
```

```
142   elif singleton (ys) :
143     splitPt = len(xs) // 2
144     (us,vs) = (xs[:splitPt],xs[splitPt:])
145     return (us, reverseF (vs))
146   else :
147     return (xs,ys[1:])
```

- The implementations are designed to satisfy the six equations:
- The equations are expressed here in Haskell notation

```
1   # -- The implementation satifies the following
2   # --
3   # -- (cons x . fromSL) ps == (fromSL . consSL x) ps
4   # -- (snoc x . fromSL) ps == (fromSL . snocSL x) ps
5   # -- (tail . fromSL) ps   == (fromSL . tailSL) ps
6   # -- (init . fromSL) ps   == (fromSL . initSL) ps
7   # -- (head . fromSL) ps   == headSL ps
8   # -- (last . fromSL) ps   == lastSL ps
```

- Each of the operations apart from `tailSL` and `initSL` take constant time
- `tailSL` and `initSL` can take linear time in the worst case but they take amortised constant time — see the references for derivation
- Note that Haskell `Data.Sequence` uses 2-3 Finger Trees for better performance
- **Ex (1)** Write down all the ways "abcd" can be represented as a symmetric list.

  Give examples to show how each of these representations can be generated.
- **Ex (2)** Define `lengthSL`
- **Ex (3)** Implement `dropWhileSL` so that

```
# dropWhile . fromSL = fromSL . dropWhileSL
```

- **Ex (4)** Define `initsSL` with the type

```
# initsSL :: SymList a -> SymList (SymList a)
```

  Write down the equation which expresses the relationship between `fromSL`, `initsSL`, and `inits`.
- **Ans (1)** There are three ways:

```
("a","dcb"),("ab","dc"),("abc","d")
```

```
Python3>>> prs1 = consSL('a',([],[]))
Python3>>> prs1
(['a'], [])
Python3>>> prs2 = snocSL('b',prs1)
Python3>>> prs2
(['a'], ['b'])
Python3>>> prs3 = snocSL('c',prs2)
Python3>>> prs3
(['a'], ['c', 'b'])
Python3>>> prs4 = snocSL('d',prs3)
Python3>>> prs4
(['a'], ['d', 'c', 'b'])

Python3>>> prs1a = snocSL('a',([],[]))
Python3>>> prs1a
([], ['a'])
Python3>>> prs2a = snocSL('b',prs1a)
Python3>>> prs2a
(['a'], ['b'])
```

- **Ans (1)** There are three ways:

```
("a","dcb"),("ab","dc"),("abc","d")
```

```
Python3>>> prs1 = consSL('d',([],[]))
Python3>>> prs1
(['d'], [])
Python3>>> prs2 = consSL('c',prs1)
Python3>>> prs2
(['c'], ['d'])
Python3>>> prs3 = consSL('b',prs2)
Python3>>> prs3
(['b', 'c'], ['d'])
Python3>>> prs4 = consSL('a',prs3)
Python3>>> prs4
(['a', 'b', 'c'], ['d'])
```

- Functional programmers will spot that the first is an instance of a `foldl` while the third is an instance of a `foldr`

10 Future Work

Programming, Debugging, Psychology

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

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

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

  A little learning is a dangerous thing;

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

Sorting, Searching, Binary Trees

- Recursive function definitions
- Inductive data type definitions

  - A *list* is either an empty list or a first item followed by the rest of the list
  - A *binary tree* is either an empty tree or a node with an item and two sub-trees
- Recursive definitions often easier to find than iterative
- Sorting
- Searching
- Both use binary tree structure
- 9 December 2021 TMA01
- Sunday 9 January 2022 Tutorial Online Sorting
- Sunday 6 February 2022 Tutorial Online Binary Trees
- 8 March 2022 TMA02

11 Example Algorithm Design — Haskell

Binary Search — Haskell

- The notes following give two implementations of *Binary Search* in Haskell
- Note: these are not part of M269 and are purely for comparison for those interested
- The first is a direct translation of the recursive Python version
- The second is derived from http://rosettacode.org/wiki/Binary_search and is more idiomatic Haskell
- The code for both implementations is in the file M269BinarySearch.hs (which should be near the file of these slides)

11.1 Binary Search — Haskell — version 1

Binary Search — Haskell — 1 (a)

```
1 module M269BinarySearch where
2
3   import Data.Array
4   import Data.List
```

- A Haskell script starts with a module header which starts with the reserved identifier, module followed by the module name, M269BinarySearch
- The module name must start with an upper case letter and is the same as the file name (without its extension of *.hs* or *.lhs*)
- Haskell uses *layout* (or the *off-side rule*) to determine scope of definitions, similar to Python
- The body of the module follows the reserved identifier where and starts with import declarations
- This imports the libraries Data.List, Data.Array

Binary Search — Haskell — 1 (b)

```
6  binarySearch :: Ord a => [a] -> a -> Maybe Int

8  binarySearch xs val
9    = binarySearch01 xs val (lo,hi)
10     where
11     lo = 0
12     hi = length xs - 1
```

- Line 8 is the definition of `binarySearch`
- The preceding line, 6, is the *type signature*
- `binarySearch` takes a list and a value of type `a` (in the class `Ord` for ordering) and returns a `Maybe Int` — `a` is a *type variable*
- The `Maybe a` type is an *algebraic data type* which is the union of the *data constructors* `Nothing` and `Just a`

```
data Maybe a = Nothing | Just a
```

Code Description 1

- `f :: t` is a *type signature* for variable `f` that reads *`f` is of type `t`*
- `f :: t1 -> t2` means that `f` has the type of a function that takes elements of type `t1` and returns elements of type `t2`
- The *function type arrow* `->` associates to the right
  - `f :: t1 -> t2 -> t3` means
  - `f :: t1 -> (t2 -> t3)`
- `f x` — function application is denoted by juxtaposition and is more binding than (almost) any other operation.
- *Function application* is left associative
  - `f x y` means
  - `(f x) y`

Binary Search — Haskell — 1 (c)

```
14 binarySearch01 :: Ord a
15   => [a] -> a -> (Int, Int) -> Maybe Int

17 binarySearch01 xs val (lo,hi)
18   = if  hi < lo then Nothing
19     else
20       let mid = (lo + hi) `div` 2
21           guess = xs !! mid
22       in
23       if val == guess
24         then Just mid
25       else if val < guess
26         then binarySearch01 xs val (lo,mid-1)
27       else   binarySearch01 xs val (mid + 1, hi)
```

Code Description 2

- A `let` expression has the form

```
    let decls in expr
```

- *decls* is a number of *declarations*
- *expr* is an expression (which is the scope of the declarations)
- `div` is the integer division function
- In `` `div` ``, the grave accents (`` ` ``) make a function into an infix operator (OK, that is syntactic sugar I need not have introduced — and my formatting program has coerced the grave accent to a left single quotation mark Unicode U+2018, not the grave accent U+0060)
- `(!!)` is the list index operator — first item has index 0

11.2 Binary Search — Haskell — version 2

Binary Search — Haskell — 2 (a)

```
29  binarySearchGen :: Integral a
30    => (a -> Ordering) -> (a, a) -> Maybe a
31  binarySearchGen p (lo,hi)
32    | hi < lo = Nothing
33    | otherwise =
34        let mid = (lo + hi) `div` 2 in
35        case p mid of
36          LT -> binarySearchGen p (lo, mid - 1)
37          GT -> binarySearchGen p (mid + 1, hi)
38          EQ -> Just mid
```

Code Description 3

- A `case` expression has the form

```
    case expr of alts
```

  expr is evaluated and whichever alternative of *alts* matches is the result

- The lines starting with `(|)` are *guarded* definitions — if the boolean expression to the right is `True` then the following expression is used
- `otherwise` is a synonym for `True`
- A *conditional* expression has the form

```
    if expr then expr else expr
```

  The first *expr* must be of type `Bool`

- *Guards* and *conditionals* are alternative styles in programming

Binary Search — Haskell — 2 (b)

```haskell
binarySearchArray :: (Ix i, Integral i, Ord a)
                     => Array i a -> a -> Maybe i
binarySearchArray ary x
  = binarySearchGen p (bounds ary)
    where
    p m = x `compare` (ary ! m)

binarySearchList :: Ord a
                 => [a] -> a -> Maybe Int
binarySearchList xs val
  =  binarySearchGen p (0, length xs - 1)
     where
     p m = val `compare` (xs !! m)
```

Code Description 4

- `compare` is a method of the `Ord` class, for *ordering*, defined in the standard *Prelude*

```haskell
class (Eq a) => Ord a where
  compare              :: a -> a -> Ordering
  (<),(<=),(>=),(>)    :: a -> a -> Bool
  max, min             :: a -> a -> a

  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT

data Ordering = LT | EQ | GT
  deriving (Eq,Ord,Enum,Read,Show,Bounded)
```

- Minimal type-specific definitions required are `compare` or `(==)` and `(<=)`
- `!` and `!!` are the array and list indexing operators

11.3 Binary Search — Haskell — Comparison

- The first version with `binarySearch` and `binarySearch01` is very similar to the *Python* recursive version `binarySearchRec`
- In the *Haskell* case an explicit helper function is used
- The second version is more general: `binarySearchGen` can be used with any type that is indexed by a data type in the `Integral` class
- `binarySearchArray` and `binarySearchList` specialise the function to arrays or lists.
- For the Haskell `Array` data type see the Haskell Report
- Idiomatic Haskell tends to be more general and make use of higher order functions, type classes and advanced features.

12 Web Links & References

- Python Online IDEs
  - **Repl.it** `https://repl.it/languages/python3` (Read-eval-print loop)

  - **TutorialsPoint CodingGround** Python 3 https://www.tutorialspoint.com/execute_python3_online.php
  - **TutorialsPoint CodingGround** Haskell ghci https://www.tutorialspoint.com/compile_haskell_online.php

- The *offside rule* (using layout to determine the start and end of code blocks) comes originally from Landin (1966) — see Off-side rule for other programming languages that use this.
- The *step-by-step* approach to writing programs is described in Glaser et al. (2000)
- The difficulty in learning programs is described in many articles — see, for example, Dehnadi and Bornat (2006)
- **Inductive data type**
  - Algebraic data type composite type — possibly recursive sum type of product types — common in modern functional languages.
  - Recursive data type from Type theory

References

Bentley, Jon (1984). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. ISSN 0001-0782. doi:10.1145/358234.381162. URL http://doi.acm.org/10.1145/358234.381162.

Bentley, Jon (1986). *Programming Pearls*. Addison Wesley. ISBN 0201103311.

Bentley, Jon (2000). *Programming Pearls*. Addison Wesley, second edition. ISBN 0201657880.

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

Bird, Richard (2010). *Pearls of Functional Algorithm Design*. Cambridge University Press. ISBN 0521513383.

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 Jeremy Gibbons (2020). *Algorithm Design with Haskell*. Cambridge University Press. ISBN 9781108869041. URL http://www.cs.ox.ac.uk/publications/books/adwh/.

Dehnadi, Saeed and Richard Bornat (2006). The camel has two humps. Web (Last checked 22 October 2015). URL http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf.

Gibbons, Jeremy (2008). Unfolding abstract datatypes. In *Mathematics of Program Construction*. Springer. doi:10.1007/978-3-540-70594-9_8. URL http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/adt.pdf.

Glaser, H; P J Hartel; and P W Garratt (2000). Programming by numbers: a programming method for complete novices. *The Computer Journal*, 43(4):252–265. A functional approach to learning programming.

Goguen, J A; J W Thatcher; E G Wagner; and J B Wright (1977). Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95.

Gries, David (1981). *The Science of Programming*. Springer. ISBN 0387964800. URL https://www.cs.cornell.edu/gries/July2016/The-Science-Of-Programming-Gries-038790641X.pdf.

Gries, David (1982). A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2(3):207–214.

Gries, David (1989). The maximum-segment-sum problem. In *Formal development programs and proofs*, pages 33–36. Addison-Wesley Longman Publishing Co., Inc.

Guttag, John (1977). Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404.

Guttag, John (1980). Notes on type abstraction (version 2). *Software Engineering, IEEE Transactions on*, 6(1):13–23.

Guttag, John V. and James J. Horning (1978). The algebraic specification of abstract data types. *Acta informatica*, 10(1):27–52.

Guttag, John V; Ellis Horowitz; and David R Musser (1978). Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064.

Hood, Robert T and Robert C Melville (1980). Real time queue operations in pure Lisp. Technical report, Cornell University.

Hoogerwoord, Rob R (1992). Functional pearls a symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513.

Jacobs, Bart and Jan Rutten (1997). A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259.

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

Liskov, Barbara and Stephen Zilles (1974). Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59.

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.

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.

Mu, Shin-Cheng (2008). Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 31–39. ACM.

Okasaki, Chris (1995). Simple and efficient purely functional queues and deques. *Journal of functional programming*, 5(04):583–592.

Okasaki, Chris (1998). *Purely Functional Data Structures*. Cambridge University Press. ISBN 0-521-63124-6.

Rutten, Jan JMM (2000). Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80.

van Rossum, Guido and Fred Drake (2003a). *An Introduction to Python*. Network Theory Limited. ISBN 0954161769.

van Rossum, Guido and Fred Drake (2003b). *The Python Language Reference Manual*. Network Theory Limited. ISBN 0954161785.

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.