M269 End of Module

M269 Tutorial 07

Phil Molyneux

3 May 2026

M269 End of Module Tutorial

Agenda

- Welcome & Introductions
- Topics from TMA03
- Abstract Data Types — Bags
- Abstract Data Types — Graphs
- Complexity
- Computability

M269 Tutorial

Introductions — Me

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

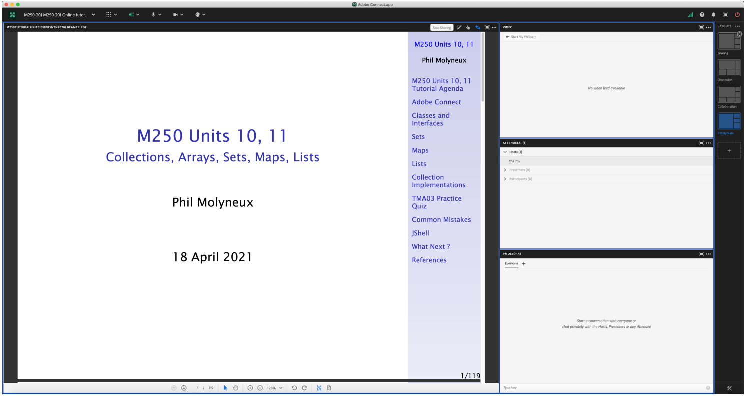
M269 Tutorial

Introductions — You

- *Name* ?
- *Position in M269* ? Which part of which Units and/or Reader have you read ?
- *Particular topics you want to look at* ?
- *Learning Syle* ?

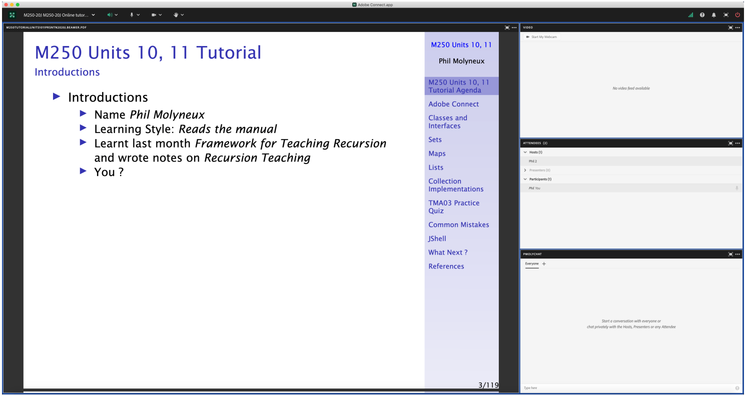
Adobe Connect

Interface — Host View



Adobe Connect

Interface — Participant View



Adobe Connect

Settings

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

Adobe Connect

Access

- **Tutor Access**

  TutorHome 〉 M269 Website 〉 Tutorials

  Cluster Tutorials 〉 M269 Online tutorial room

  Tutor Groups 〉 M269 Online tutor group room

  Module-wide Tutorials 〉 M269 Online module-wide room

- **Attendance**

  TutorHome 〉 Students 〉 View your tutorial timetables

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

  Attendee Pod 〉 Menu 〉 Clear Everyone's Status

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

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

- **Presenter Only Area**

  Meeting 〉 Enable/Disable Presenter Only Area

Adobe Connect

Keystroke Shortcuts

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

Adobe Connect Interface

Sharing Screen & Applications

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

Adobe Connect

Ending a Meeting

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

Adobe Connect

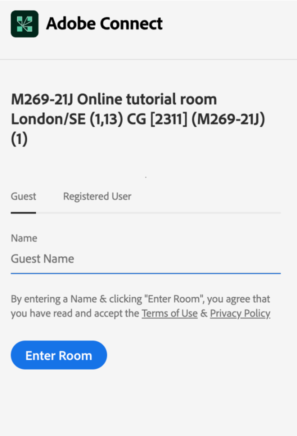
Invite Attendees

- **Provide Meeting URL** Menu › Meeting › Manage Access & Entry › Invite Participants. . .
- **Allow Access without Dialog** Menu › Meeting › Manage Meeting Information provides new browser window with *Meeting Information* Tab bar › Edit Information
- Check *Anyone who has the URL for the meeting can enter the room*
- Default *Only registered users and accepted guests may enter the room*
- **Reverts to default next session but URL is fixed**
- Guests have blue icon top, registered participants have yellow icon top — same icon if URL is open
- See Start, attend, and manage Adobe Connect meetings and sessions

Adobe Connect

Entering a Room as a Guest (1)

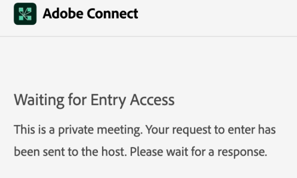
- Click on the link sent in email from the Host
- Get the following on a Web page
- As *Guest* enter your name and click on Enter Room



Adobe Connect

Entering a Room as a Guest (2)

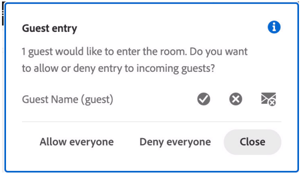
- See the *Waiting for Entry Access* for *Host* to give permission



Adobe Connect

Entering a Room as a Guest (3)

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



Adobe Connect

Layouts

- **Creating new layouts** example *Sharing* layout
- Menu › Layouts › Create New Layout. . . › Create a New Layout dialog › Create a new blank layout and name it *PMolyMain*
- New layout has no Pods but does have Layouts Bar open (see Layouts menu)
- **Pods**
- Menu › Pods › Share › Add New Share and resize/position — initial name is *Share n* — rename *PMolyShare*
- **Rename Pod** Menu › Pods › Manage Pods. . . › Manage Pods › Select › Rename or Double-click & rename
- Add Video pod and resize/reposition
- Add Attendance pod and resize/reposition
- Add Chat pod — rename it *PMolyChat* — and resize/reposition

Adobe Connect

Layouts

- Dimensions of **Sharing** layout (on 27-inch iMac)
  - Width of Video, Attendees, Chat column 14 cm
  - Height of Video pod 9 cm
  - Height of Attendees pod 12 cm
  - Height of Chat pod 8 cm
- **Duplicating Layouts** does *not* give new instances of the Pods and is probably not a good idea (apart from local use to avoid delay in reloading Pods)
- **Auxiliary Layouts** name *PMolyAux0n*
  - Create new Share pod
  - Use existing Chat pod
  - Use same Video and Attendance pods

Adobe Connect

Chat Pods

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

Graphics Conversion

PDF to PNG/JPG

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

Adobe Connect Recordings

Exporting Recordings

- *Menu bar* › *Meeting* › Preferences › Video
- Aspect ratio › Standard (4:3) (not Wide screen (16:9) default)
- Video quality › Full HD (1080p not High default 480p)
- **Recording** *Menu bar* › *Meeting* › Record Session ✔
- **Export Recording**
- *Menu bar* › *Meeting* › Manage Meeting Information
- *New window* › Recordings › *check Tutorial* › Access Type button
- check Public › check Allow viewers to download
- **Download Recording**
- *New window* › Recordings › *check Tutorial* › Actions › Download File

M269 25J TMA03
Topics

- Part 1 Qs 1 – 2
- Part 2 Qs 3 –
- Q1 Recursion and trees
- Q2 Backtracking
- Q3 Complexity classes
- Q4 Turing machine
- Q5 Computability

Backtracking Notes

Introduction

- These notes are sketches on Backtracking for a sequence of notes on exhaustive searching
- The notes draw on:
- M269 25J Book Chp 22 *Backtracking*
- Bob Moore Backtracking tutorial slides and Python code
- Bird and Wadler (1988, page 161) *Introduction to Functional Programming*
- Bird and Gibbons (2020, sec 15.1) *Implicit search and the n-queens problem*

  Both of the *Bird* references have some discussion of efficiency and improving performance — you should be aware the the former constructs partial solutions column (File) by column while the later constructs partial solution row (Rank) by row

Searching

Exhaustive, Greedy, Backtracking

- **Exhaustive** look at every option which fits the basic criteria
- **Greedy** take locally optimum choice at every stage

  Hardly ever works — requires a proof that this approach works for this particular case

  Proofs not often included in a course — unfortunate

  When greedy *does* work, it is more efficient than other methods
- **Backtracking** check if a partial solution can be extended towards a full solution

  If not, go back and make a different choice

  The *backtracking* may be done implicitly via recursion

Backtracking

Template

- **Problem** choice of representation and initial setup — define *partial solutions*
- Define global constraints that must be satisfied by any solution (but perhaps not by partial solutions)
- Define local constraints that must be satisfied by partial solutions and determines if a partial solution can be extended
- Note that backtracking relies on local contraints — if there are no local constraints then you are on an exhaustive search
- **extend** — checks for solution
- **satisfiesGlobal** checks if solution satisfies global constraints
- **canExtend** checks if can extend partial solution

Backtracking

8-Queens Problem (1)

- The problem of placing eight queens on an 8x8 chessboard so that no two queens threaten each other.
- Problem posed by Max Bezzel, chess composer, in 1848
- First solution by Franz Nauck in 1850
- Carl Friedrich Gauss also worked on the 8-queens problem and the generalised *n*-queens problem
- In 1874 S Günther proposed a solution method using determinants. J W L Glaisher refined the approach
- Edsger Dijkstra 1972 used 8 queens to illustrate structured programming and backtracking

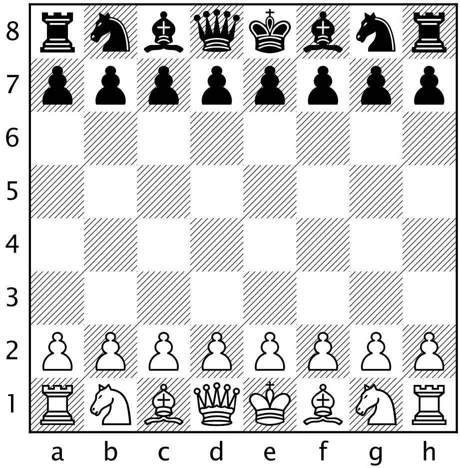
Backtracking

8-Queens Problem (2)

- Each column (or row) must contain exactly one queen. So a strategy for finding a solution is as follows.
- Place a queen in in the first column in any position.
- Then place a queen in the second column in any position not in check from the first queen.
- Continue until all eight queens have been placed.
- If at any point this is not possible (because all positions are in check) then *backtrack* and reapply the method to find a different position for the queens in the first m columns and try again

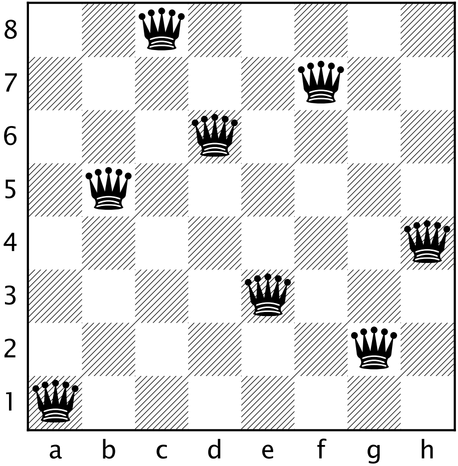
Backtracking

8-Queens Problem (3)



Backtracking

8-Queens Problem (4)



Backtracking

8-Queens Problem (5)

- **Board representation** list of ranks (rows)
- The board above is `[1,5,8,6,3,7,2,4]`
- **Health warning** some implementations represent a board as list of files (columns) and may count from top to bottom
- These notes are based on:
- Bird and Wadler (1988, sec 6.5.1 page 161) *Introduction to Functional Programming*
- Bird and Gibbons (2020, sec 15.1 page 369) uses ranks (rows) not files (columns)
- Bob Moore's M269 Backtracking tutorial notes 2026
- M269 notes chp 22
- Knuth (2023, sec 7.2.2 page 30) *The Art of Computer Programming: The Combinatorial Aldorithms Volume 4B*

Backtracking

8-Queens Problem (6)

- We first declare some type aliases for position and chessboard

```
type Rank  = int    # Rows
type File  = int    # Columns
type Board = [Rank] # Example
```

Backtracking

8-Queens Problem (7)

- The function isSafe(brd,n) tests whether the partial solution brd can be extended by one column with a queen in Rank *n*
- isSafe(brd,n) uses the function inCheck(posn1,posn2) to test if two queens at posn1 and posn2 hold each other in check

```python
def inCheck(posn1: (File,Rank), posn2: (File,Rank)) -> bool :
  """ Return True if Queen in position posn1 == (i,j)
  can be threatened by Queen in posn2 == (m.n)
  Precondition: not (i == m)
  since we are placing queens File (column) by File
  """
  (i,j) = (posn1[0],posn1[1])
  (m,n) = (posn2[0],posn2[1])
  return ((j == n)
          or (i + j == m + n)
          or (i - j == m - n))

def isSafe(brd: Board, n: Rank) -> bool :
  nxtFl = len(brd) + 1
  lstOfCks = [not (inCheck((i,j), (nxtFl,n)))
              for (i,j) in list(zip(range(1,len(brd)+1), brd))
              ]
  return all(lstOfCks)
```

Backtracking

8-Queens Problem (8)

- The function queens takes a board size brdSize and recursively finds all partial solutions of size m
- queeens8 and queens4 specialise queens to boards of size 8 and 4

```
def queens(brdSize: Rank,m: File) -> [Board] :
  if m == 0 :
    return [[]]
  else :
    lstBrd = [brd + [n]
              for brd in queens(brdSize, m-1)
              for n in range(1,brdSize + 1)
              if isSafe(brd,n)]
    return lstBrd


def queens8(m: File) -> [Board] :
  return queens(8, m)

def queens4(m: File) -> [Board] :
  return queens(4, m)
```

Backtracking

4-Queens Example (1)

- Knuth (2023, sec 7.2.2 page 31) *Backtrack Programming* suggests that one of the best ways to learn about backtracking is to execute the algorithm by hand in the special case of `queens(4,4)`
- This is done with the chessboard illustrated and colour coded
  - Blue for partial solutions
  - Red for not feasible
  - Green for solutions
- The initial blank board is not shown
- Following the boards is a different visualisation of the the *backtracking tree* with some further results for boards of different sizes

Backtracking

4-Queens Example (2)

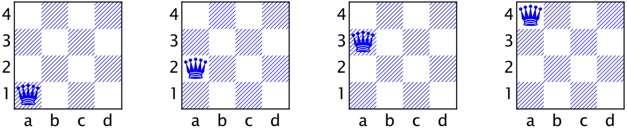
- Working column by column, fill the next column where the position is safe

```
Python3>>> queens4(0)
[[]]                                # by line 33
Python3>>> queens4(1)
[[1], [2], [3], [4]]                # by line 35
Python3>>> queens4(2)               # by line 35
[[1, 3], [1, 4], [2, 4], [3, 1], [4, 1], [4, 2]]
Python3>>> queens4(3)               # by line 35
[[1, 4, 2], [2, 4, 1], [3, 1, 4], [4, 1, 3]]
Python3>>> queens4(4)
[[2, 4, 1, 3], [3, 1, 4, 2]]        # by line 35
```

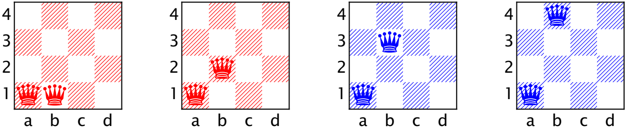
Backtracking

4-Queens Example (3)

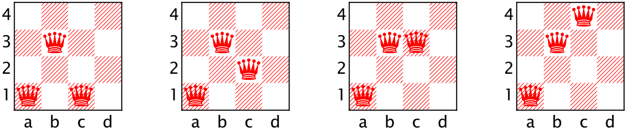
- Level 1



- Level 2 Block 1



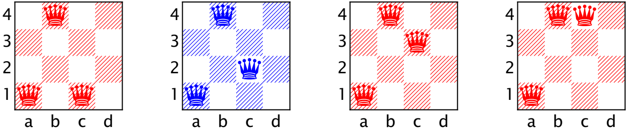
- Level 3 Block 1



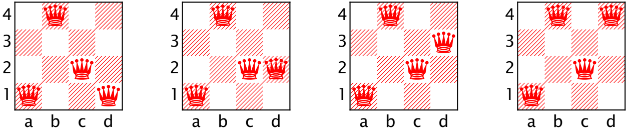
Backtracking

4-Queens Example (4)

- Level 3 Block 2



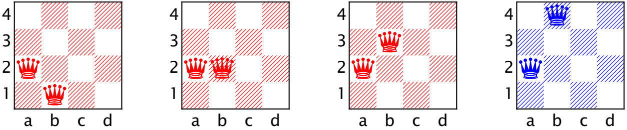
- Level 4 Block 1



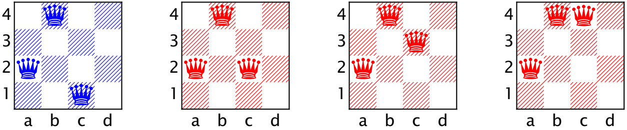
Backtracking

4-Queens Example (5)

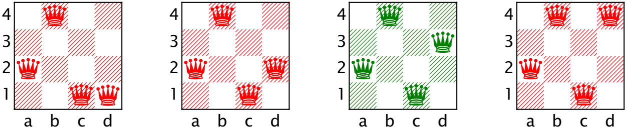
- Level 2 Block 2



- Level 3 Block 3



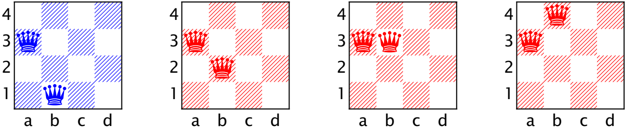
- Level 4 Block 2



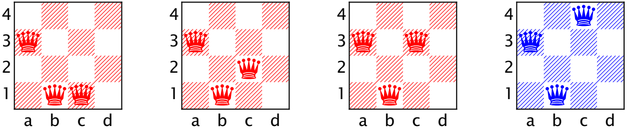
Backtracking

4-Queens Example (6)

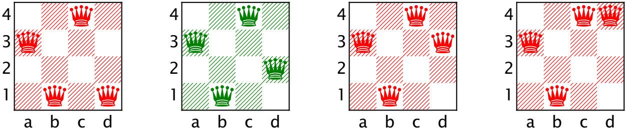
- Level 2 Block 3



- Level 3 Block 4



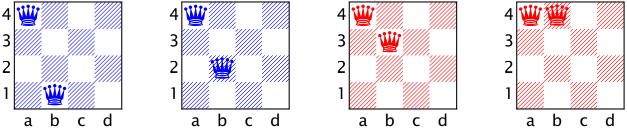
- Level 4 Block 3



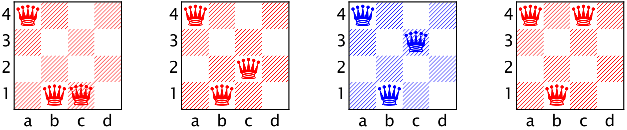
Backtracking

4-Queens Example (7)

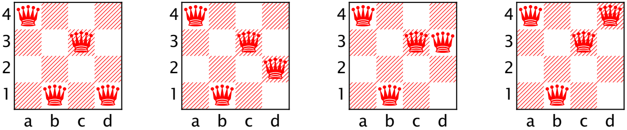
- Level 2 Block 4



- Level 3 Block 5



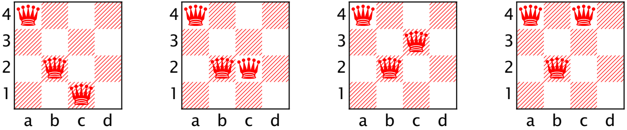
- Level 4 Block 4



Backtracking

4-Queens Example (8)

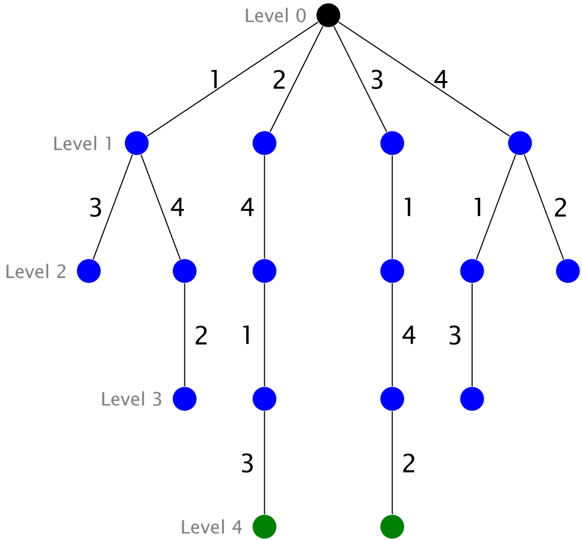
- Level 3 Block 6



- The above chessboards *can* be used to form a tree of queen placement on the board
- This can also be represented as a *Backtrack tree* — see below

Backtracking

4-Queens Example — Backtrack Tree



Backtracking

4-Queens Example — Profiles

- The *profile* $(p_0, p_1, \ldots, p_n)$ of a backtrack tree is the number of nodes at each level

```python
def backtrackProfile(brdSize: Rank) -> list[int] :
  return [len(queens(brdSize,m)) for m in range(brdSize+1)]
```

```
Python3>>> backtrackProfile(4)
[1, 4, 6, 4, 2]
Python3>>> sum(backtrackProfile(4))
17
Python3>>> backtrackProfile(8)
[1, 8, 42, 140, 344, 568, 550, 312, 92]
Python3>>> sum(backtrackProfile(8))
2057
```

- **queens4** Solutions 2
  - Backtrack nodes 17
  - Number of possible sequences $4^4 = 256$
  - Arbitrary placings $\binom{16}{4} = 1820$
- **queens8** Solutions 92
  - Backtrack nodes 2057
  - Number of possible sequences $8^8 = 16{,}777{,}216$
  - Arbitrary placings $\binom{64}{8} = 4{,}426{,}165{,}368$

Backtracking

n-Queens Solutions

#-Qns	# Solns	#-Qns	# Solns
0	1	14	365,596
1	1	15	2,279,184
2	0	16	14,772,512
3	0	17	95,815,104
4	2	18	666,090,624
5	10	19	4,968,057,848
6	4	20	39,029,188,884
7	40	21	314,666,222,712
8	92	22	2,691,008,701,644
9	352	23	24,233,937,684,440
10	724	24	227,514,171,973,736
11	2680	25	2,207,893,435,808,352
12	14,200	26	22,317,699,616,364,044
13	73,712	27	234,907,967,154,122,528

- **Source** OEIS: A000170

Backtracking

n-Queens Solutions

- Knuth (2023, page 33) gives some anecdotes about the *n*-Queens calculations
- $Q(11)$ and $Q(12)$ were the last to be calculated by hand, the latter in 1910
- Q(13) was calculated using a computer in 1960
- The calculation of $Q(14)$ was done in 1963 taking 120 hours on an IBM 1620
- $Q(15)$ took two hours in 1974 on an IBM 360-75
- $Q(27)$ was calculated in 2016 taking 383 days using a cluster of FPGA devices — it will probably be some time before this is exceeded

Backtracking

n-Queens Solutions

- Here are the timings and memory usage for my home computer — an Apple M1 Max from 2022 with 32 GB memory
- The code is a Haskell translation of Python listed earlier

```
GHCi> :set +s
GHCi> backtrackProfileSum(11)
(166926,[1,11,90,536,2468,8492,21362,37248,44148,34774,15116,2680])
(23.28 secs, 27,516,106,792 bytes)
GHCi> backtrackProfileSum(12)
(856189,[1,12,110,756,4080,16852,52856,120104
        ,195270,222720,160964,68264,14200])
(148.38 secs, 174,226,075,008 bytes)
```

```haskell
backtrackProfile :: Rank -> [Int]
backtrackProfile brdSize
  = [length (queens brdSize m) |
     m <- [0..brdSize]]

backtrackProfileSum :: Rank -> (Int,[Int])
backtrackProfileSum brdSize
  = (profileSum,profileLst)
    where
      profileLst = backtrackProfile brdSize
      profileSum = sum profileLst
```

Backtracking

n-Queens Solutions

- Here are further timings and memory usage for my home computer
- This is for sizes 13 and 14
- Notice we are heading towards some serious waiting even with not very large inputs

```
GHCi> :set +s
GHCi> backtrackProfileSum(13)
(4674890,[1,13,132,1030,6404,31100,117694
         ,335010,707698,1086568,1151778,813448,350302,73712])
(998.87 secs, 1,152,159,900,192 bytes)
GHCi> backtrackProfileSum(14)
(27358553,[1,14,156,1364,9632,54068,241484,835056
          ,2211868,4391988,6323032,6471872,4511922,1940500,365596])
(7317.33 secs, 8,046,531,055,416 bytes)
```

Backtracking

8-Queens Problem — References

- Bird and Wadler (1988, sec 6.5.1 page 161) *Introduction to Functional Programming*
- Bird and Gibbons (2020, sec 15.1 page 369) *Algorithm Design with Haskell*
- Knuth (2023, sec 7.2.2 page 31) *Backtrack Programming*
- Bell and Brett (2009) *A survey of known results and research areas for n-queens*

Backtracking

8-Queens Problem — References

- Rosetta Code: N-queens problem
- Chess.com: Eight Queens Puzzle
- Medium: Eight Queens Problem (members only)
- OEIS: All solutions to the problem of eight queens
- OEIS: n Queens
- John D Cook: Special solutions to the eight queens problem
- Using Symmetry to Optimize an N-Queens Counting Algorithm
- How 8 Queens Works
- Solving 8-queens with determinants comment by Mike Spivey
- Backtracking Algorithms
- N Queen Problem

Bags

Definitions

- **Bag** unordered collection that may contain duplicate items
- Also known as Multiset
- **Multiplicity** of an element is the number of instances of an element
- A Bag or Multiset may be defined as a two-tuple (A, m) where A is the underlying set from which elements are drawn, and a function $m : A \to \mathbb{N}$
- Note that some definitions exclude 0 from the range of m so it would be denoted $m : A \to \mathbb{N}_{\geqslant 1}$

Bags

Example

- 120 has prime factorization $120 = 2^3 \times 3^1 \times 5^1$
- Gives bag $\{2, 2, 2, 3, 5\}$
- The $\{\}$ is an abuse of the usual set notation
- Representations:
- Sorted list $[2, 2, 2, 3, 5]$
- Unsorted list $[2, 3, 2, 5, 2]$
- Sorted list of pairs $[(2, 3), (3, 1), (5, 1)]$

Bags

Operations

- **Create** an empty bag
- **Queries**
  - `isEmptyBag`
  - `sizeBag` total number of elements
  - `elemOccurs` number of an element
  - `elemMember` is there at least one copy of an element
- **Construction**
  - `insertElem` insert one copy of an element
  - `insertMany` insert a number of copies
  - `deleteElem` delete a single copy of an element
  - `deleteMany` delete a number of copies
  - `deleteAll` deleteAll all copies

Bags

M269 2018J TMA02 Bag Operations

- `Bag()` creates a new empty bag
- `add(self, elem)` adds one copy of `elem` to the bag `self`
- `count(self,elem)` number of `elem` in the bag `self`
- `size(self)` total number of copies in `self`
- `clear(self,elem)` removes all copies of `elem`
- `ordered(self)` returns contents of bag `self` as a list of pairs `(count, elem)` in decreasing order of `count`

Bags
Implementations

- Python Counter is a subclass of dict which is Python's version of bags or multisets
- Data.MultiSet is Haskell's implementation of multisets or bags — it is based on Data.Map
- Multiset describes the ADT multiset or bag in several programming languages

Python `dict`

Creation

- A mapping object or dictionary maps hashable values to arbitrary objects
- A dictionary is a mutable object
- **Creation**

```
aD = dict()      # dictionary constructor
aD = {}          # literal empty dictionary
aD = {'to': 2, 'be': 2, 'or': 1, 'not': 1}
                 # literal key:value pairs
```

Python dict

Creation

- Creation methods

```
Python3>>> aD = {'to': 2, 'be': 2
...            , 'or': 1, 'not': 1}
Python3>>> bD = dict(to=2, be=2, orA=1, notA=1)
Python3>>> cD = dict(zip(['to','be','or','not']
...                    ,[2,2,1,1]))
Python3>>> dD = dict([('to',2),('be',2)
...                  ,('or',1),('not',1)])
Python3>>> eD = dict({'to': 2, 'be': 2
...                  , 'or': 1, 'not': 1})
Python3>>> aD == cD == dD == eD
True
```

- Keywords in keyword arguments must not clash with built-in keywords
- Implicit line joining means we can split expressions in parentheses, square brackets or curly braces over more than one physical line without using backslashes.

Python dict
Queries

- Queries in the *M269 Companion*

```
Python3>>> aD = {'to': 2, 'be': 2
...            , 'or': 1, 'not': 1}
Python3>>> len(aD)
4
Python3>>> key = 'be'
Python3>>> key in aD
True
Python3>>> aD[key]
2
Python3>>> aD.keys()
dict_keys(['to', 'or', 'not', 'be'])
Python3>>> list(aD.keys())
['to', 'or', 'not', 'be']
Python3>>> aD.items()
dict_items([('to', 2), ('or', 1)
           , ('not', 1), ('be', 2)])
Python3>>> list(aD.items())
[('to', 2), ('or', 1), ('not', 1), ('be', 2)]
Python3>>> ('to',2) in aD.items()
True
```

Dictionaries

Activity 1 Total Exercise

- Write a function, `totalValue`, that takes a dictionary, `xD`, and returns the total of all the values of the key:value pairs (Assumes value is a numeric type)

Go to Answer

- Answer 1 Total Exercise

```
def totalValue(xD) :
  """
  totalValue takes a dictionary, xD,
  and returns the total of all the values
  of the key:value pairs
  Assumes value is a numeric type
  """

  tVal = sum([v for (k,v) in xD.items()])
  return tVal

  # Alternative
  # tVal = sum(xD.values())
```

- Answer 1 continued on next slide

Go to Activity

Dictionaries

Answer 1 Total Exercise (contd)

- Answer 1 Total Exercise — sample use

```
Python3>>> aD
{'not': 1, 'be': 2, 'to': 2, 'or': 1}
Python3>>> totalValue(aD)
6
```

Go to Activity

Python dict

Modifiers

- Modifiers in the *M269 Companion*

```
Python3>>> aD = {'to': 2, 'be': 2
...            , 'or': 1, 'not': 1}
Python3>>> aD
{'not': 1, 'be': 2, 'to': 2, 'or': 1}
Python3>>> aD['dobe'] = 2
Python3>>> aD
{'not': 1, 'be': 2, 'dobe': 2, 'to': 2, 'or': 1}
Python3>>> aD['do'] = 1
Python3>>> aD
{'do': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 2, 'dobe': 2}
Python3>>> aD['be'] = 3
Python3>>> aD
{'do': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3, 'dobe': 2}
```

Python `dict`

Modifiers (2)

- Modifiers in the *M269 Companion*

```
Python3>>> aD
{'do': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3, 'dobe': 2}
Python3>>> aD['do'] = aD['do'] + 1
Python3>>> aD
{'do': 2, 'to': 2, 'or': 1
, 'not': 1, 'be': 3, 'dobe': 2}
Python3>>> del aD['dobe']
Python3>>> aD
{'do': 2, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
```

Python dict

Errors

- It is an error to access a non-existant key

```
Python3>>> aD
{'do': 2, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
Python3>>> aD['dobe']
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
KeyError: 'dobe'
Python3>>> del aD['dobe']
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
KeyError: 'dobe'
Python3>>> aD['dobe'] = aD['dobe'] + 1
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
KeyError: 'dobe'
```

Python dict

Errors (2)

- If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary. Not an error but could catch you out

```
Python3>>> fD = {'one' : 2, 'one': 3}
Python3>>> fD
{'one': 3}
```

Dictionaries

Activity 2 Add One Exercise

- Write a function, addOneToKey, that takes a key, key, a dictionary, xD, and adds 1 to the value of the key

▸ Go to Answer

Dictionaries

Answer 2 Add One Exercise

- Answer 2 Add One Exercise

```python
def addOneToKey(key, xD) :
  """
  addOneToKey takes a key and a dictionary
  and adds one to the value of the key
  Invariant for xD:
    if key in xD :
      xD[key] > 0
  """

  if key in xD :
    xD[key] = xD[key] + 1
  else :
    xD[key] = 1

  return xD
```

- Answer 2 continued on next slide

Go to Activity

Dictionaries

Answer 2 Add One Exercise (contd)

- Answer 2 Add One Exercise
- Note that addOneToKey has a side effect on the input dictionary

```
Python3>>> aD
{'do': 2, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
Python3>>> addOneToKey('do',aD)
{'do': 3, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
Python3>>> aD
{'do': 3, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
Python3>>> addOneToKey('abba',aD)
{'do': 3, 'abba': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
Python3>>> aD
{'do': 3, 'abba': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
```

Go to Activity

Dictionaries

Activity 3 Delete One Exercise

- Write a function, `delOneFromKey`, that takes a key, `key`, a dictionary, `xD`, and subtracts 1 from the value of the key if the key is in `xD`

Go to Answer

Delete One Exercise

Answer 3 Delete One Exercise

- Answer 3 Delete One Exercise

```python
def delOneFromKey(key, xD) :
  """
  delOneFromKey takes a key and a dictionary
  and deletes one from the value of the key
  Invariant for xD:
    if key in xD :
      xD[key] > 0
  """

  if not key in xD :
    pass
  elif xD[key] == 1 :
    del xD[key]
  else :
    xD[key] = xD[key] - 1

  return xD
```

- Answer 3 continued on next slide

Go to Activity

Delete One Exercise

Answer 3 Delete One Exercise (contd)

- Answer 3 Delete One Exercise — examples

```
Python3>>> aD
{'do': 3, 'abba': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
Python3>>> delOneFromKey('do',aD)
{'do': 2, 'abba': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
Python3>>> aD
{'do': 2, 'abba': 1, 'to': 2, 'or': 1
, 'not': 1, 'be': 3}
Python3>>> delOneFromKey('abba',aD)
{'do': 2, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
Python3>>> aD
{'do': 2, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
Python3>>> delOneFromKey('bbc',aD)
{'do': 2, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
Python3>>> aD
{'do': 2, 'to': 2, 'or': 1, 'not': 1, 'be': 3}
```

Go to Activity

Python Counter

Overview, Creation

- Counter objects are part of the collections library for container datatypes
- A Counter is a subclass of the mapping type dict and has a dictionary interface with some differences and extensions
- Creation:

```
#!/usr/bin/env python3

from collections import Counter
```

```
AnPython3>>> ct1 = Counter()
AnPython3>>> ct1              # new empty Counter
Counter()
AnPython3>>> ct2 = Counter('lambda calculus')
AnPython3>>> ct2              # new Counter from iterable
Counter({'l': 3, 'a': 3, 'c': 2, 'u': 2, 'm': 1, 'b': 1, 'd': 1, ' ': 1, 's': 1})
```

Python Counter

Creation

```
AnPython3>>> ct3 = Counter(a=10, b=0, c=-2, d=4)
AnPython3>>> ct3               # Counter from keyword args
Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
AnPython3>>> ct4 = {'a': 10, 'd': 4, 'b': 0, 'c': -2}
AnPython3>>> ct4               # not a Counter
{'a': 10, 'd': 4, 'b': 0, 'c': -2}
AnPython3>>> type(ct4)
<class 'dict'>
AnPython3>>> ct5 = Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
AnPython3>>> ct5               # Counter from a mapping
Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
```

Python Counter

Access, and Operations

```
AnPython3>>> ct5 = Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
AnPython3>>> ct5
Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
AnPython3>>> ct5['e']
0
AnPython3>>> ct6 = +ct5      # remove zero and negative elements
AnPython3>>> ct6
Counter({'a': 10, 'd': 4})
AnPython3>>> ct7 = -ct5
AnPython3>>> ct7
Counter({'c': 2})
```

- Accessing missing elements is not an error (unlike a dictionary)
- Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

Python Counter

Views

```
AnPython3>>> ct5 = Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
AnPython3>>> ct5
Counter({'a': 10, 'd': 4, 'b': 0, 'c': -2})
AnPython3>>> ct8 = ct5.items()
AnPython3>>> ct8
dict_items([('a', 10), ('d', 4), ('b', 0), ('c', -2)])
AnPython3>>> ct9 = ct5.keys()
AnPython3>>> ct9
dict_keys(['a', 'd', 'b', 'c'])
AnPython3>>> ct10 = ct5.values()
AnPython3>>> ct10
dict_values([10, 4, 0, -2])
```

- `items()` Return a new view of the dictionary's items (`(key, value)` pairs).
- `keys()` Return a new view of the dictionary's keys.
- `values()` Return a new view of the dictionary's values.
- See Dictionary view objects
- They provide a dynamic view on the dictionary's entries
- When he dictionary changes, the view changes as well

M269 Bags

Prsntn 2021J TMA03 Bags

```python
from collections import Counter

"""
Implementation of Bag ADT for M269 Prsntn 2021J TMA03 Q1
"""

class Bag :

  def __init__(self) :
    """Create a new empty bag."""
    self.items = Counter()
    self.count = 0

  def size(self) -> int:
    """Return the total number of copies of all items in the bag."""
    return self.count
```

M269 Bags

Prsntn 2021J TMA03 Bags

```
def add(self, item: object) -> None :
  """Add one copy of item to the bag.
     Multiple copies are allowed."""
  self.items[item] =  self.items[item] + 1
  self.count = self.count + 1

def discard(self, item: object) -> None :
  """ Remove at most one copy of item from the bag.
      No effect if item is not in the bag.
  """
  if self.items[item] > 0 :
    self.items[item] =  self.items[item] - 1
    self.count = self.count - 1
```

M269 Bags

Prsntn 2021J TMA03 Bags

```python
def contains(self, item: object) -> bool :
  """ Return True if there is at least
      one copy of item in the bag.
  """
  # Add your own code here to replace the following statement
  pass
```

- **Hint** what can a `Counter` do that in a `dict` would generate an error ?

M269 Bags

Prsntn 2021J TMA03 Bags

```python
def multiplicity(self, item: object) -> int :
  """Return the number of copies of item in the bag.
  Return zero if the item doesn’t occur in the bag.
  """
  # Add your own code here to replace the following statement
  pass
```

- **Hint** remember that `Counter` is a subclass of `dict`

M269 Bags

Prsntn 2021J TMA03 Bags

```
def ordered(self) :
  """Return the items ordered by decreasing multiplicity.
  Return a list of (count, item) pairs.
  """
  # You will be asked to add your own code here later
  pass
```

- **Hint** Modify the list comprehension below to only include items with count > 0
- What functions are available to sort a list ?

```
AnPython3>>> ct8 = ct5.items()
AnPython3>>> ct8
dict_items([('a', 10), ('d', 4), ('b', 0), ('c', -2)])
AnPython3>>> ct11 = [(ct,itm) for (itm,ct) in ct8]
AnPython3>>> ct11
[(10, 'a'), (4, 'd'), (0, 'b'), (-2, 'c')]
```

- See Sorting HOW TO
- What is the difference between sorted() and list.sort() ?

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

Overview (2)

- **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

Abstract Data Types

Overview (3)

- **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

Abstract Data Types

Overview (4) — Shapes ADT Style

```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

Abstract Data Types

Overview (5) — Shapes OO Style

```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)
```

Abstract Data Types

Overview (6)

- Haskell Type Classes are similar to Java/OOP Interfaces
- See OOP vs type classes
- See Java's Interface and Haskell's type class: differences and similarities?
- See Difference between OOP interfaces and FP type classes
- See What exactly makes the Haskell type system so revered (vs say, Java)?
- **Health Warning** Much of OO programming is using the *OO syntax* to create ADTs

Commentary 2

Computability

1 Computability

- Description of Turing Machine
- Turing Machine examples
- Computability, Decidability and Algorithms
- Non-computability — Halting Problem
- Reductions and non-computability
- Lambda Calculus (optional)
- Note that the Computability notes are here mainly for reference since the Complexity notes refer to them
- This session is mainly on the Complexity topics

Computability

Ideas of Computation

- The idea of an algorithm and what is effectively computable
- **Church-Turing thesis** Every function that would naturally be regarded as computable can be computed by a deterministic Turing Machine. (Unit 7 Section 4)
- See Phil Wadler on computability theory performed as part of the Bright Club at The Strand in Edinburgh, Tuesday 28 April 2015

Computability

Models of Computation

- In automata theory, a *problem* is the question of deciding whether a given string is a member of some particular language
- If Σ is an alphabet, and L is a language over Σ, that is $L \subseteq \Sigma^*$, where Σ^* is the set of all strings over the alphabet Σ then we have a more formal definition of *decision problem*
- Given a string $w \in \Sigma^*$, decide whether $w \in L$
- Example: Testing for a prime number — can be expressed as the language L_p consisting of all binary strings whose value as a binary number is a prime number (only divisible by 1 or itself)
- See Hopcroft (2007, section 1.5.4)

Automate Theory

Alphabets, Strings

- An **Alphabet**, Σ, is a finite, non-empty set of symbols.
- Binary alphabet $\Sigma = \{0, 1\}$
- Lower case letters $\Sigma = \{a, b, \ldots, z\}$
- A **String** is a finite sequence of symbols from some alphabet
- 01101 is a string from the Binary alphabet $\Sigma = \{0, 1\}$
- The **Empty string**, ϵ, contains no symbols
- **Powers**: Σ^k is the set of strings of length k with symbols from Σ
- The set of all strings over an alphabet Σ is denoted Σ^*
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- **Question** Does $\Sigma^0 = \varnothing$? ($\varnothing$ is the empty set)

Automata Theory

Languages

- An **Language**, L, is a subset of Σ^*
- The set of binary numerals whose value is a prime
  $\{10, 11, 101, 111, 1011, \ldots\}$
- The set of binary numerals whose value is a square
  $\{100, 1001, 10000, 11001, \ldots\}$

Computability

Church-Turing Thesis & Quantum Computing

- **Church-Turing thesis** Every function that would naturally be regarded as computable can be computed by a deterministic Turing Machine.
- **physical Church-Turing thesis** Any finite physical system can be simulated (to any degree of approximation) by a Universal Turing Machine.
- **strong Church-Turing thesis** Any finite physical system can be simulated (to any degree of approximation) with polynomial slowdown by a Universal Turing Machine.
- Shor's algorithm (1994) — quantum algorithm for factoring integers — an NP problem that is not known to be P — also not known to be NP-complete and we have no proof that it is not in P
- Reference: Section 4 of Unit 6 & 7 Reader

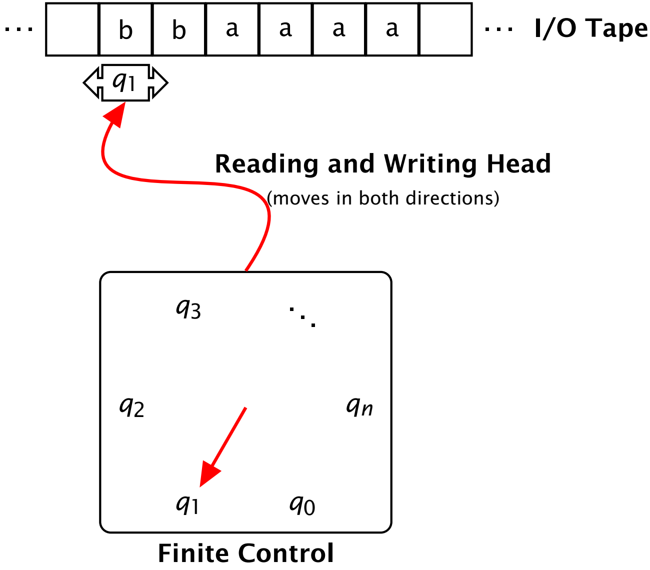
Computability

Turing Machine

- **Finite control** which can be in any of a finite number of *states*
- **Tape** divided into cells, each of which can hold one of a finite number of symbols
- Initially, the **input**, which is a finite-length string of symbols in the *input alphabet*, is placed on the tape
- All other tape cells (extending unbounded left and right) hold a special symbol called *blank*
- A **tape head** which initially is over the leftmost input symbol
- A **move** of the Turing Machine depends on the state and the tape symbol scanned
- A move can change state, write a symbol in the current cell, move left, right or stay
- References: Hopcroft (2007, page 326), Unit 6 & 7 Reader (section 5.3)

Turing Machine Diagram

Turing Machine Diagram



Computability

Turing Machine notation

- Q finite set of states of the finite control
- Σ finite set of *input symbols* (M269 S)
- Γ complete set of *tape symbols* $\Sigma \subset \Gamma$
- δ Transition function (M269 instructions, I)
  $\delta :: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
- $\delta(q, X)$ takes a state, q and a tape symbol, X and returns (p, Y, D) where p is a state, Y is a tape symbol to overwrite the current cell, D is a direction, Left, Right or Stay
- q_0 *start state* $q_0 \in Q$
- B *blank symbol* $B \in \Gamma$ and $B \notin \Sigma$
- F set of *final or accepting states* $F \subseteq Q$

Turing Machine Examples

Turing Machine Simulators

- Morphett's Turing machine simulator — the examples below are adapted from here
- Ugarte's Turing machine simulator
- XKCD A Bunch of Rocks — XKCD Explanation

  Image below (will need expanding to be readable)
- The term *state* is used in two different ways:

  The value of the *Finite Control*

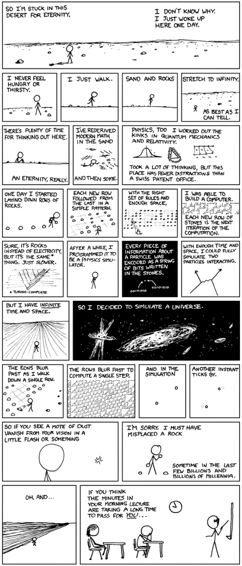
  The overall configuration of *Finite Control* and current contents of the tape

  See Turing Machine: State

  will lead to some confusion

Turing Machine Examples

XKCD A Bunch of Rocks



Turing Machine Examples

Meta-Exercise

- For each of the Turing Machine Examples below, identify

  $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Turing Machine Examples

The Successor Function

- **Input** binary representation of numeral n
- **Output** binary representation of $n + 1$
- Example $1010 \mapsto 1011$ and $1011 \mapsto 1100$
- Initial cell: leftmost symbol of n
- **Strategy**
- **Stage A** make the rightmost cell the current cell
- **Stage B** Add 1 to the current cell.
- If the current cell is 0 then replace it with 1 and go to stage C
- If the current cell is 1 replace it with 0 and go to stage B and move Left
- If the current cell is blank, replace it by 1 and go to stage C
- **Stage C** Finish up by making the leftmost cell current

Turing Machine Examples

The Successor Function (2)

- Represent the Turing Machine program as a list of quintuples (q, X, p, Y, D)
- **Stage A**

  $(q_0, 0, q_0, 0, R)$

  $(q_0, 1, q_0, 1, R)$

  (q_0, B, q_1, B, L)
- **Stage B**

  $(q_1, 0, q_2, 1, S)$

  $(q_1, 1, q_1, 0, L)$

  $(q_1, B, q_2, 1, S)$
- **Stage C**

  $(q_2, 0, q_2, 0, L)$

  $(q_2, 1, q_2, 1, L)$

  (q_2, B, q_h, B, R)

Turing Machine Examples

The Successor Function (2a)

- **Exercise** Translate the quintuples (q, X, p, Y, D) into English and check they are the same as the specification
- **Stage A** make the rightmost cell the current cell

  $(q_0, 0, q_0, 0, R)$

  If state q_0 and read symbol 0 then stay in state q_0 write 0, move R

  $(q_0, 1, q_0, 1, R)$

  If state q_0 and read symbol 1 then stay in state q_0 write 1, move R

  (q_0, B, q_1, B, L)

  If state q_0 and read symbol B then state q_1 write B, move L

Turing Machine Examples

The Successor Function (2b)

- **Exercise** Translate the quintuples (q, X, p, Y, D) into English
- **Stage B** Add 1 to the current cell.

  $(q_1, 0, q_2, 1, S)$

  If state q_1 and read symbol 0 then state q_2 write 1, stay

  $(q_1, 1, q_1, 0, L)$

  If state q_1 and read symbol 1 then state q_1 write 0, move L

  $(q_1, B, q_2, 1, S)$

  If state q_1 and read symbol B then state q_2 write 1, stay

Turing Machine Examples

The Successor Function (2c)

- **Exercise** Translate the quintuples (q, X, p, Y, D) into English
- **Stage C** Finish up by making the leftmost cell current

  $(q_2, 0, q_2, 0, L)$

  If state q_2 and read symbol 0 then state q_2 write 0, move L

  $(q_2, 1, q_2, 1, L)$

  If state q_2 and read symbol 1 then state q_2 write 0, move L

  (q_2, B, q_h, B, R)

  If state q_2 and read symbol B then state q_h write B, move R HALT
- Notice that the Turing Machine feels like a series of `if ... then` or `case` statements inside a `while` loop

Turing Machine Examples

The Successor Function (2d) — Meta-Exercise

- Identify $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Turing Machine Examples

The Successor Function (2e) — Meta-Exercise

- Identify $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$
- $Q = \{q_0, q_1, q_2, q_h\}$
- q_0 finding the rightmost symbol
- q_1 add 1 to current cell
- q_2 move to leftmost cell
- q_h finish
- $\Sigma = \{0, 1\}$
- $\Gamma = \Sigma \cup \{B\}$
- $\delta :: Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
  δ is represented as {(q,X,p,Y,D)}
  equivalent to $\{((q, X), (p, Y, D))\}$ *set of pairs*
- q_0 start with leftmost symbol under head, state moving to rightmost symbol
- B is ␣ a visible space
- $F = \{q_h\}$

Turing Machine Examples

The Successor Function (3)

- **Sample Evaluation** $11 \mapsto 100$
- **Representation** $\cdots BX_1 X_2 \cdots X_{i-1}\, qX_iX_{i+1} \cdots X_nB \cdots$

  $q_0 11$

  $1 q_0 1$

  $11 q_0 B$

  $1 q_1 1$

  $q_1 10$

  $q_1 B00$

  $q_2 100$

  $q_2 B100$

  $q_h 100$
- **Exercise** evaluate $1011 \mapsto 1100$

Turing Machine Examples

Instantaneous Description

- **Representation** $\cdots BX_1X_2 \cdots X_{i-1}\,qX_iX_{i+1} \cdots X_nB \cdots$
- q is the state of the TM
- The head is scanning the symbol X_i
- Leading or trailing blanks B are (usually) not shown unless the head is scanning them
- $\vdash_M$ denotes one move of the TM M
- $\vdash_M^*$ denotes zero or more moves
- $\vdash$ will be used if the TM M is understood
- If (q, X_i, p, Y, L) denotes a TM move then
  $X_1 \cdots X_{i-1}\,qX_i \cdots X_n \vdash_M X_1 \cdots X_{i-2}\,pX_{i-1}\,Y \cdots X_n$

Turing Machine Examples

The Binary Palindrome Function

- **Input** binary string s
- **Output** YES if palindrome, NO otherwise
- Example $1010 \mapsto NO$ and $1001 \mapsto YES$
- Initial cell: leftmost symbol of s
- **Strategy**
- **Stage A** read the leftmost symbol
- If blank then accept it and go to stage D otherwise erase it
- **Stage B** find the rightmost symbol
- If the current cell matches leftmost recently read then erase it and go to stage C
- Otherwise reject it and go to stage E
- **Stage C** return to the leftmost symbol and stage A
- **Stage D** print YES and halt
- **Stage E** erase the remaining string and print NO

Turing Machine Examples

The Binary Palindrome Function (2)

- Represent the Turing Machine program as a list of quintuples (q, X, p, Y, D)
- **Stage A** read the leftmost symbol

  $(q_0, 0, q_{1_o}, B, R)$

  $(q_0, 1, q_{1_i}, B, R)$

  (q_0, B, q_5, B, S)
- **Stage B** find rightmost symbol

  $(q_{1_o}, B, q_{2_o}, B, L)$

  $(q_{1_o}, *, q_{1_o}, *, R)$ * is a wild card, matches anything

  $(q_{1_i}, B, q_{2_i}, B, L)$

  $(q_{1_i}, *, q_{1_i}, *, R)$

Turing Machine Examples

The Binary Palindrome Function (3)

- **Stage B** check

  $(q_{2_o}, 0, q_3, B, L)$

  (q_{2_o}, B, q_5, B, S)

  $(q_{2_o}, *, q_6, *, S)$

  $(q_{2_i}, 1, q_3, B, L)$

  (q_{2_i}, B, q_5, B, S)

  $(q_{2_i}, *, q_6, *, S)$

- **Stage C** return to the leftmost symbol and stage A

  (q_3, B, q_5, B, S)

  $(q_3, *, q_4, *, L)$

  (q_4, B, q_0, B, R)

  $(q_4, *, q_4, *, L)$

Turing Machine Examples

The Binary Palindrome Function (4)

- **Stage D** accept and print YES

  $(q_5, *, q_{5_a}, Y, R)$

  $(q_{5_a}, *, q_{5_b}, E, R)$

  $(q_{5_b}, *, q_7, S, S)$

- **Stage E** erase the remaining string and print NO

  (q_6, B, q_{6_a}, N, R)

  $(q_6, *, q_6, B, L)$

  $(q_{6_a}, *, q_7, O, S)$

- Finish

  (q_7, B, q_h, B, R)

  $(q_7, *, q_7, *, L)$

Turing Machine Examples

The Binary Palindrome Function (3a) — Meta-Exercise

- Identify $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Turing Machine Examples

The Binary Palindrome Function (3b) — Meta-Exercise

- Identify $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$
- $Q = \{q_0, q_{1_o}, q_{1_i}, q_{2_o}, q_{2_i}, q_3, q_4, q_5, q_{5_a}, q_{5_b}, q_6, q_{6_a}, q_7, q_h\}$
- q_0 read leftmost symbol
- q_{1_o}, q_{1_i} find rightmost symbol looking for 0 or 1
- q_{2_o}, q_{2_i} check, confirm or reject
- q_3, q_4 check finish or move to start
- q_5, q_6, q_7 print YES or NO and finish
- q_h finish
- $\Sigma = \{0, 1\}$
- $\Gamma = \Sigma \cup \{B, Y, E, S, N, O\}$
- $\delta :: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
  δ is represented as {(q,X,p,Y,D)}
  equivalent to $\{((q, X), (p, Y, D))\}$ *set of pairs*
- Start with leftmost symbol under head, state q_0
- B is ␣ a visible space
- $F = \{q_h\}$

Turing Machine Examples

The Binary Palindrome Function (4)

- **Sample Evaluation** 101 $\mapsto$ YES

  $q_0 101 \vdash B q_{1_i} 01 \vdash B0 q_{1_i} 1 \vdash B01 q_{1_i} B$

  $\vdash B0 q_{2_i} 1$

  $\vdash B q_3 0B \vdash q_4 B0B$

  $\vdash B q_0 0B \vdash BB q_{1_o} B$

  $\vdash B q_{2_o} BB$

  $\vdash B q_5 BB \vdash Y q_{5_a} B \vdash YE q_{5_b} B \vdash YE q_7 S$

  $\vdash Y q_7 ES \vdash B q_7 YES \vdash q_7 BYES \vdash q_h YES$
- **Exercise** Evaluate 110 $\mapsto$ NO

Turing Machine Examples

Binary Addition Example

- **Input** two binary numerals separated by a single space $n1\ n2$
- **Output** binary numeral which is the sum of the inputs
- Example $110110 + 101011 \mapsto 1100001$
- Initial cell: leftmost symbol of $n1\ n2$
- **Insight** look at the arithmetic algorithm

␣	1	1	0	1	1	0
␣	1	0	1	0	1	1
1	1	0	0	0	0	1

- **Discussion** how can we overwrite the first number with the result and remember how far we have gone ?

Turing Machine Examples

Binary Addition Example — Arithmetic Reinvented

␣	1	1	0	1	1	0
␣	1	0	1	0	1	1
—	—	—	—	—	—	—
␣	1	1	0	1	1	y
␣	1	0	1	0	1	␣
—	—	—	—	—	—	—
␣	1	1	1	0	x	y
␣	1	0	1	0	␣	␣
—	—	—	—	—	—	—
␣	1	1	1	x	x	y
␣	1	0	1	␣	␣	␣
—	—	—	—	—	—	—
1	0	0	x	x	x	y
␣	1	0	␣	␣	␣	␣
—	—	—	—	—	—	—
1	0	x	x	x	x	y
␣	1	␣	␣	␣	␣	␣
—	—	—	—	—	—	—
1	y	x	x	x	x	y
␣	␣	␣	␣	␣	␣	␣
—	—	—	—	—	—	—
1	1	0	0	0	0	1

Turing Machine Examples

Binary Addition Example (2)

- **Input** two binary numerals separated by a single space
  $n1\ n2$
- **Output** binary numeral which is the sum of the inputs
- Example $110110 + 101011 \mapsto 1100001$
- Initial cell: leftmost symbol of $n1\ n2$
- **Strategy**
- **Stage A** find the rightmost symbol

  If the symbol is 0 erase and go to stage Bx

  If the symbol is 1 erase go to stage By

  If the symbol is blank go to stage F

  dealing with each digit in $n2$

  if no further digits in $n2$ go to final stage
- **Stage Bx** Move left to a blank go to stage Cx
- **Stage By** Move left to a blank go to stage Cy

  moving to $n1$

Turing Machine Examples

Binary Addition Example (3)

- **Stage Cx** Move left to find first 0, 1 or B

  Turn 0 or B to X, turn 1 to Y and go to stage A

  adding 0 to a digit finalises the result (no *carry one*)
- **Stage Cy** Move left to find first 0, 1 or B

  Turn 0 or B to 1 and go to stage D

  Turn 1 to 0, move left and go to stage Cy

  dealing with the *carry one* in school arithmetic
- **Stage D** move right to X, Y or B and go to stage E
- **Stage E** replace 0 by X, 1 by Y, move right and go to Stage A

  finalising the value of a digit resulting from a *carry*
- **Stage F** move left and replace X by 0, Y by 1 and at B halt

Turing Machine Examples

Binary Addition Example (4)

- Represent the Turing Machine program as a list of quintuples (q, X, p, Y, D)
- **Stage A** find the rightmost symbol

  (q_0, B, q_1, B, R)

  $(q_0, *, q_0, *, R)$ * is a wild card, matches anything

  (q_1, B, q_2, B, L)

  $(q_1, *, q_1, *, R)$

  $(q_2, 0, q_{3_x}, B, L)$

  $(q_2, 1, q_{3_y}, B, L)$

  (q_2, B, q_7, B, L)

Turing Machine Examples

Binary Addition Example (5)

- **Stage Bx** move left to blank

  $(q_{3_x}, B, q_{4_x}, B, L)$

  $(q_{3_x}, *, q_{3_x}, *, L)$

- **Stage By** move left to blank

  $(q_{3_y}, B, q_{4_y}, B, L)$

  $(q_{3_y}, *, q_{3_y}, *, L)$

- **Stage Cx** move left to 0, 1, or blank

  $(q_{4_x}, 0, q_0, x, R)$

  $(q_{4_x}, 1, q_0, y, R)$

  (q_{4_x}, B, q_0, x, R)

  $(q_{4_x}, *, q_{4_x}, *, L)$

Turing Machine Examples

Binary Addition Example (6)

- **Stage Cy** move left to 0, 1, or blank

  $(q_{4_y}, 0, q_5, 1, S)$

  $(q_{4_y}, 1, q_{4_y}, 0, L)$

  $(q_{4_y}, B, q_5, 1, S)$

  $(q_{4_y}, *, q_{4_y}, *, L)$

- **Stage D** move right to x, y or B

  (q_5, x, q_6, x, L)

  (q_5, y, q_6, y, L)

  (q_5, B, q_6, B, L)

  $(q_5, *, q_5, *, R)$

- **Stage E** replace 0 by x, 1 by y

  $(q_6, 0, q_0, x, R)$

  $(q_6, 1, q_0, y, R)$

Turing Machine Examples

Binary Addition Example (7)

- **Stage F** replace x by 0, y by 1

  $(q_7, x, q_7, 0, L)$

  $(q_7, y, q_7, 1, L)$

  (q_7, B, q_h, B, R)

  $(q_7, *, q_7, *, L)$

- **Exercise** Evaluate $11 + 10 \mapsto 101$

Turing Machine Examples

The Binary Addition Function (7a) — Meta-Exercise

- Identify $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Turing Machine Examples

The Binary Addition Function (7b) — Meta-Exercise

- Identify $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$
- $Q = \{q_0, q_1, q_2, q_{3_x}, q_{3_y}, q_{4_x}, q_{4_y}, q_5, q_6, q_7, q_h\}$
- q_0, q_1, q_2 find rightmost symbol of second number
- q_{3_x}, q_{3_y} move left to inter-number blank
- q_{4_x}, q_{4_y} move left to 0, 1 or blank
- q_5 move right to x, y or B
- q_6 replace 0 by x, 1 by y and move right
- q_7 replace x by 0, y by 1 and move left
- q_h finish
- $\Sigma = \{0, 1\}$
- $\Gamma = \Sigma \cup \{B, x, y\}$
- $\delta :: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
  δ is represented as $\{(q, X, p, Y, D)\}$
  equivalent to $\{((q, X), (p, Y, D))\}$ *set of pairs*
- Start with leftmost symbol under head, state q_0
- B is ␣ a visible space
- $F = \{q_h\}$

Turing Machine Examples

Binary Addition Example (8a)

- **Exercise** Evaluate $11 + 10 \mapsto 101$
- **Stage A** find the rightmost symbol

  $BBq_0 11B10B$ Note space symbols B at start and end

$\vdash BB1q_0 1B10B$

$\vdash BB11q_0 B10B$

$\vdash BB11Bq_1 10B$

$\vdash BB11B1q_1 0B$

$\vdash BB11B10q_1 B$

$\vdash BB11B1q_2 0B$

$\vdash BB11Bq_{3_x} 1BB$

- **Stage Bx** move left to blank

$\vdash B11q_{3_x} B1BB$

- **Stage Cx** move left to 0, 1, or blank

$\vdash BB1q_{4_x} 1B1BB$

$\vdash BB1Yq_0 B1BB$

Turing Machine Examples

Binary Addition Example (8b)

- **Exercise** Evaluate $11 + 10 \mapsto 101$ (contd)
- **Stage A** find the rightmost symbol

$\vdash$ $BB1BYBq_1 1BB$

$\vdash$ $BB1YB1q_1 BB$

$\vdash$ $BB1YBq_2 1BB$

$\vdash$ $BB1Yq_{3_y} BBBB$

- **Stage Cy** move left to 0, 1, or blank

$\vdash$ $BB1q_{4_y} YBBBB$

$\vdash$ $BBq_{4_y} 1YBBBB$

$\vdash$ $Bq_{4_y} B0YBBBB$

$\vdash$ $Bq_5 10YBBBB$

- **Stage D** move right to x, y or B

$\vdash$ $Bq_5 0YBBBB$

$\vdash$ $B0q_5 YBBBB$

$\vdash$ $Bq_6 0YBBBB$

- **Stage E** replace 0 by x, 1 by y

$\vdash$ $B1Xq_0 YBBBB$

Turing Machine Examples

Binary Addition Example (8c)

- **Exercise** Evaluate $11 + 10 \mapsto 101$ (contd)
- **Stage A** find the rightmost symbol

$\vdash B1XYq_0BBBB$

$\vdash B1XYBq_1BBB$

$\vdash B1XYq_2BBBB$

$\vdash B1Xq_7YBBBB$

- **Stage F** replace x by 0, y by 1

$\vdash B1q_7X1BBBB$

$\vdash Bq_7101BBBB$

$\vdash Bq_7B101BBBB$

$\vdash Bq_h101BBBB$

- This is mimicking what you learnt to do on paper as a child! Real *step-by-step* instructions
- See Morphett's Turing machine simulator for more examples (takes too long by hand!)

Computability

Universal Turing Machine

- Universal Turing Machine, U, is a Turing Machine that can simulate any arbitrary Turing machine, M
- Achieves this by encoding the transition function of M in some standard way
- The input to U is the encoding for M followed by the data for M
- See Turing machine examples

Computability

Decidability

- **Decidable** — there is a TM that will halt with yes/no for a decision problem — that is, given a string *w* over the alphabet of *P* the TM with halt and return yes.no the string is in the language *P* (same as *recursive* in Recursion theory — old use of the word)
- **Semi-decidable** — there is a TM will halt with yes if some string is in *P* but may loop forever on some inputs (same as *recursively enumerable*) — *Halting Problem*
- **Highly-undecidable** — no outcome for any input — *Totality, Equivalence Problems*

Computability

Undecidable Problems

- **Halting problem** — the problem of deciding, given a program and an input, whether the program will eventually halt with that input, or will run forever — term first used by Martin Davis 1952
- **Entscheidungsproblem** — the problem of deciding whether a given statement is provable from the axioms using the rules of logic — shown to be undecidable by Turing (1936) by reduction from the *Halting problem* to it
- **Type inference and type checking** in the second-order lambda calculus (important for functional programmers, Haskell, GHC implementation)
- Undecidable problem — see link to list

Computability

Halting Problem — Sketch Proof (1)

- Halting problem — is there a program that can determine if any arbitrary program will halt or continue forever ?
- Assume we have such a program (Turing Machine) `h(f,x)` that takes a program `f` and input `x` and determines if it halts or not

```
h(f,x)
  = if f(x) runs forever
      return True
    else
      return False
```

- We shall prove this cannot exist by contradiction

Computability

Halting Problem — Sketch Proof (2)

- Now invent two further programs:
- q(f) that takes a program f and runs h with the input to f being a copy of f
- r(f) that runs q(f) and halts if q(f) returns True, otherwise it loops

```
q(f)
  = h(f,f)

r(f)
  = if q(f)
      return
    else
      while True: continue
```

- What happens if we run r(r) ?
- If it loops, q(r) returns True and it does not loop — contradiction.
- Scooping theLoop Snooper: A proof that the Halting Problem is undecidable Geoffrey K Pullum (21 May 2024)

Computability

Why undecidable problems must exist

- A *problem* is really membership of a string in some language
- The number of different languages over any alphabet of more than one symbol is uncountable
- Programs are finite strings over a finite alphabet (ASCII or Unicode) and hence countable.
- There must be an infinity (big) of problems more than programs.
- **Computational problem** — defined by a function
- **Computational problem is computable** if there is a Turing machine that will calculate the function.

Computability

Computability and Terminology (1)

- The idea of an *algorithm* dates back 3000 years to Euclid, Babylonians...
- In the 1930s the idea was made more formal: which *functions* are *computable*?
- A *function* is a set of pairs $f = \{(x, f(x)) : x \in X \wedge f(x) \in Y\}$ with the *function property*
- *Function property:* $(a, b) \in f \wedge (a, c) \in f \Rightarrow b == c$
- *Function property:* Same input implies same output
- Note that maths notation is deeply inconsistent here — see Function and History of the function concept
- *What do we mean by computing a function — an algorithm ?*

Functions

Relation and Rule

- The idea of function as a set of pairs (Binary relation) with the function property (each element of the domain has at most one element in the co-domain) is fairly recent — see History of the function concept
- School maths presents us with function as rule to get from the input to the output
- Example: the square function: $\texttt{square}\ \ x = x \times x$
- But lots of rules (or algorithms) can implement the same function
- $\texttt{square1}\ \ x\ \ \texttt{=}\ \ x\texttt{\^{}2}$
- $\texttt{square2}\ \ x\ \ \texttt{=}\ \ \overbrace{x + \cdots + x}^{x\ \texttt{times}}$ if x is integer

Computability

Computability and Terminology (2)

- In the 1930s three definitions:
- λ-Calculus, simple semantics for computation — Alonzo Church
- General recursive functions — Kurt Gödel
- Universal (Turing) machine — Alan Turing
- Terminology:
  - Recursive, recursively enumerable — Church, Kleene
  - Computable, computably enumerable — Gödel, Turing
  - Decidable, semi-decidable, highly undecidable
  - In the 1930s, *computers* were *human*
  - Unfortunate choice of terminology
- Turing and Church showed that the above three were equivalent
- Church-Turing thesis — function is intuitively computable if and only if Turing machine computable

Computability

Reducing one problem to another

- To reduce problem P_1 to P_2, invent a construction that converts instances of P_1 to P_2 that have the same answer. That is:
  - any string in the language P_1 is converted to some string in the language P_2
  - any string over the alphabet of P_1 that is not in the language of P_1 is converted to a string that is not in the language P_2
- With this construction we can solve P_1
  - Given an instance of P_1, that is, given a string w that may be in the language P_1, apply the construction algorithm to produce a string x
  - Test whether x is in P_2 and give the same answer for w in P_1

Computability

Problem Reduction

- **Problem Reduction — Ordinary Example**
- Want to phone Alice but don't have her number
- You know that Bill has her number
- So *reduce* the problem of finding Alice's number to the problem of getting hold of Bill

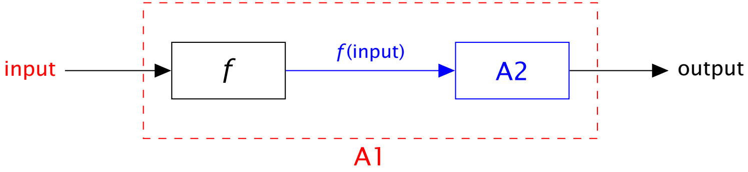
Computability

Direction of Reduction

- The direction of reduction is important
- If we can reduce P_1 to P_2 then (in some sense) P_2 is at least as hard as P_1 (since a solution to P_2 will give us a solution to P_1)
- So, if P_2 is decidable then P_1 is decidable
- To show a problem is undecidable we have to reduce from an known undecidable problem to it
- $\forall x(\mathrm{dp}_{P_1}(x) = \mathrm{dp}_{P_2}(\mathrm{reduce}(x)))$
- Since, if P_1 is undecidable then P_2 is undecidable

Reductions & Non-Computable

Reductions

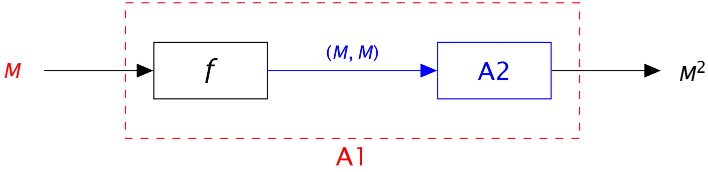


- A *reduction* of problem P_1 to problem P_2
  - transforms inputs to P_1 into inputs to P_2
  - runs algorithm $A2$ (which solves P_2) and
  - interprets the outputs from $A2$ as answers to P_1
- More formally: A problem P_1 is *reducible* to a problem P_2 if there is a function f that takes any input x to P_1 and transforms it to an input $f(x)$ of P_2

  such that the solution of P_2 on $f(x)$ is the solution of P_1 on x

Reductions & Non-Computible

Example: Squaring a Matrix



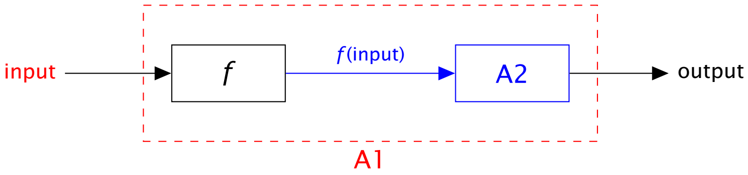
- Given an algorithm ($A2$) for matrix multiplication (P_2)
  - Input: pair of matrices, (M_1, M_2)
  - Output: matrix result of multiplying M_1 and M_2
- P_1 is the problem of squaring a matrix
  - Input: matrix M
  - Output: matrix M^2
- Algorithm $A1$ has

  $f(M) = (M, M)$

  uses $A2$ to calculate $M \times M = M^2$

Reductions & Non-Computable

Non-Computable Problems



- If P_2 is computable ($A2$ exists) then P_1 is computable (f being simple or polynomial)
- Equivalently If P_1 is non-computable then P_2 is non-computable
- **Exercise:** show $B \rightarrow A \equiv \neg A \rightarrow \neg B$

Reductions & Non-Computable

Contraposition

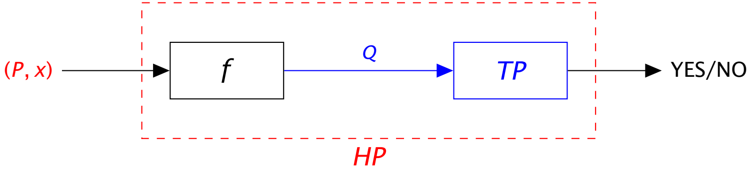
- Proof by Contrapositive
- $B \rightarrow A \equiv \neg B \vee A$ by truth table or equivalences

  $\equiv \neg(\neg A) \vee \neg B$ commutativity and negation laws

  $\equiv \neg A \rightarrow \neg B$ equivalences
- Common error: switching the order round

Reductions & Non-Computable

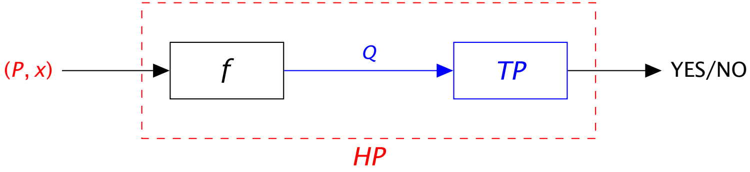
Totality Problem



- **Totality Problem**
  - Input: program Q
  - Output: YES if Q terminates for all inputs else NO
- Assume we have algorithm TP to solve the Totality Problem
- Now reduce the Halting Problem to the Totality Problem

Reductions & Non-Computable

Totality Problem



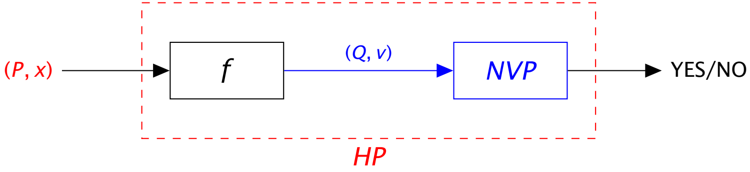
- Define f to transform inputs to HP to TP pseudo-Python

```python
def f(P,x) :
  def Q(y):
    # ignore y
    P(x)
  return Q
```

- Run *TP* on *Q*
  - If *TP* returns YES then *P* halts on *x*
  - If *TP* returns NO then *P* does not halt on *x*
- We have *solved* the Halting Problem — contradiction

Reductions & Non-Computable

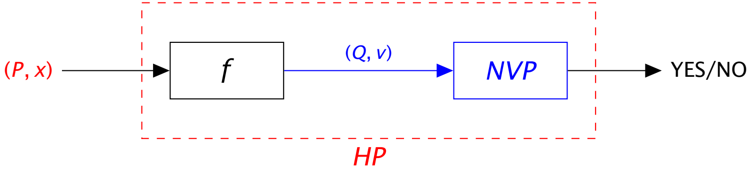
Negative Value Problem



- **Negative Value Problem**
  - Input: program Q which has no input and variable v used in Q
  - Output: YES if v ever gets assigned a negative value else NO
- Assume we have algorithm *NVP* to solve the Negative Value Problem
- Now reduce the Halting Problem to the Negative Value Problem

Reductions & Non-Computable

Negative Value Problem



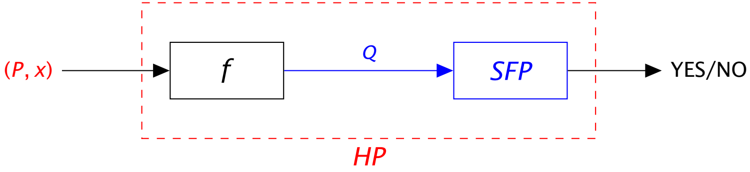
- Define f to transform inputs to HP to NVP pseudo-Python

```
def f(P,x) :
  def Q(y):
    # ignore y
    P(x)
    v = -1
  return (Q,var(v))
```

- Run NVP on $(Q, var(v))$ `var(v)` gets the variable name
  - If NVP returns YES then P halts on x
  - If NVP returns NO then P does not halt on x
- We have *solved* the Halting Problem — contradiction

Reductions & Non-Computable

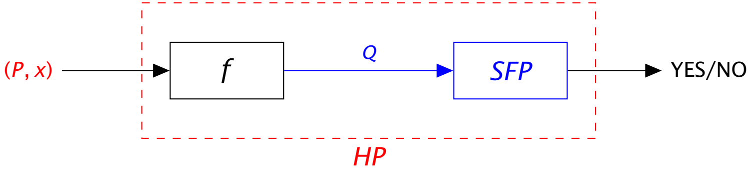
Squaring Function Problem



- **Squaring Function Problem**
  - Input: program *Q* which takes an integer, *y*
  - Output: YES if *Q* always returns the square of *y* else NO
- Assume we have algorithm *SFP* to solve the Squaring Function Problem
- Now reduce the Halting Problem to the Squaring Function Problem

Reductions & Non-Computable

Squaring Function Problem



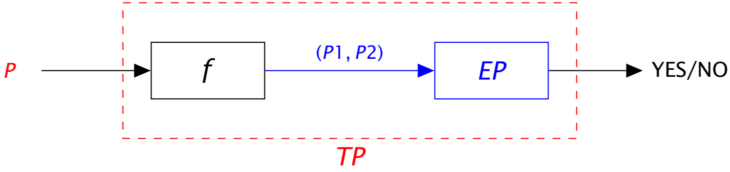
- Define *f* to transform inputs to HP to SFP pseudo-Python

```
def f(P,x) :
  def Q(y):
    P(x)
    return y * y
  return Q
```

- Run *SFP* on *Q*
  - If *SFP* returns YES then *P* halts on *x*
  - If *SFP* returns NO then *P* does not halt on *x*
- We have *solved* the Halting Problem — contradiction

Reductions & Non-Computable

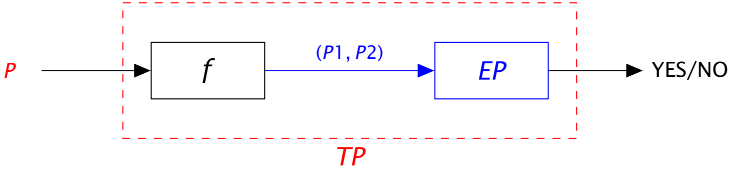
Equivalence Problem



- **Equivalence Problem**
  - Input: two programs *P*1 and *P*2
  - Output: YES if *P*1 and *P*2 solve the same problem (same output for same input) else NO
- Assume we have algorithm *EP* to solve the Equivalence Problem
- Now reduce the Totality Problem to the Equivalence Problem

Reductions & Non-Computable

Equivalence Problem



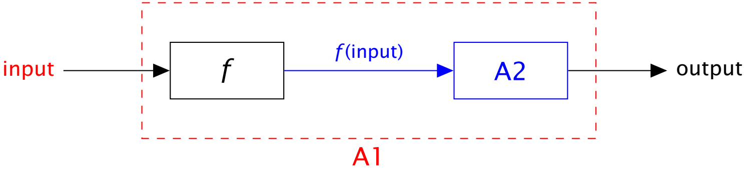
- Define *f* to transform inputs to TP to EP pseudo-Python

```
def f(P) :
  def P1(x):
    P(x)
    return "Same␣string"
  def P2(x)
    return "Same␣string"
  return (P1,P2)
```

- Run *EP* on $(P1, P2)$
  - If *EP* returns YES then *P* halts on all inputs
  - If *EP* returns NO then *P* does not halt on all inputs
- We have *solved* the Totality Problem — contradiction

Reductions & Non-Computable

Rice's Theorem



- Rice's Theorem all non-trivial, semantic properties of programs are undecidable. H G Rice 1951 PhD Thesis
- Equivalently: For any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property.
- A property of partial functions is called trivial if it holds for all partial computable functions or for none.

Reductions & Non-Computable

Rice's Theorem

- Rice's Theorem and computability theory
- Let S be a set of languages that is nontrivial, meaning
  - there exists a Turing machine that recognizes a language in S
  - there exists a Turing machine that recognizes a language not in S
- Then, it is undecidable to determine whether the language recognized by an arbitrary Turing machine lies in S.
- This has implications for compilers and virus checkers
- Note that Rice's theorem does not say anything about those properties of machines or programs that are not also properties of functions and languages.
- For example, whether a machine runs for more than 100 steps on some input is a decidable property, even though it is non-trivial.

Lambda Calculus

Motivation

- Lambda Calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution
- Lambda calculus is Turing complete — it can simulate any Turing machine
- Introduced by Alonzo Church in 1930s
- Basis of functional programming languages — Lisp, Scheme, ISWIM, ML, SASL, KRC, Miranda, Haskell, Scala, F#. . .
- **Note** this is not part of M269 but may help understand ideas of computability

Functions

Binding and Substitution

- School maths introduces functions as $f(x) = 3x^2 + 4x + 5$
- Substitution: $f(2) = 3 \times 2^2 + 4 \times 2 + 5 = 25$
- Generalise: $f(x) = ax^2 + bx + c$
- What is wrong with the following:
- $f(a) = a \times a^2 + b \times a + c$
- The ideas of free and bound variables and substitution

Expressions

Evaluation Strategies (a)

- In evaluating an expression we have choices about the order in which we evaluate subterms
- Some choices may involve more work than others but the Church-Rosser theorem ensures that if the evaluation terminates then all choices get to the same answer
- The second edition of a famous book on Functional programming — Bird (1998, Ex 1.2.2, page 6) *Introduction to Functional Programming using Haskell* — had the following exercise:
- How many ways can you evaluate $(3 + 7)^2$

  List the evaluations and assumptions
- The first edition — Bird and Wadler (1988, Ex 1.2.1, page 6) *Introduction to Functional Programming* — had the exercise:
- How many ways can you evaluate $\left((3 + 7)^2\right)^2$

Expressions

Evaluation Strategies (b)

- How many ways can you evaluate $(3+7)^2$

  List the evaluations and assumptions
- **Answer** 3 ways
- **Reducible expressions** (redexes)

  $x^2 \rightarrow x \times x$ where x is a term

  $a + b$ where a and b are numbers

  $x \times y$ where x and y are numbers

```
[sqr (3+7),((3+7)*(3+7)),((3+7)*10),(10*10),100]
[sqr (3+7),((3+7)*(3+7)),(10*(3+7)),(10*10),100]
[sqr (3+7),sqr 10,(10*10),100]
```

- The assumed redexes do not include distributive laws

  $(a+b) \times (x+y) \rightarrow a \times x + a \times y + b \times x + b \times y$
- This would increase the number of different evaluations

Expressions

Evaluation Strategies (c)

- How many ways can you evaluate $\left((3+7)^2\right)^2$
- **Answer** 547 ways

```
1 [sqr sqr (3+7),(sqr (3+7)*sqr (3+7)),(sqr (3+7)*((3+7)*(3+7))), (sqr (3+7)*((3+7)*10)
2 [sqr sqr (3+7),(sqr (3+7)*sqr (3+7)),(sqr(3+7)*((3+7)*(3+7))), (sqr (3+7)*((3+7)*10)
```

```
546 [sqr sqr (3+7),sqr sqr 10,sqr (10*10), ((10*10)*(10*10)),(100*(10*10)),(100*100),100
547 [sqr sqr (3+7),sqr sqr 10,sqr (10*10),sqr 100,(100*100),10000]
```

- Enumerating all 547 ways may have taken some concentration

Expressions

Evaluation Strategies (d)

- The actual Evaluation strategy used by a particular programming language implementation may have optimisations which make an evaluation which looks costly to be somewhat cheaper
- For example, the Haskell implementation GHC optimises the evaluation of common subexpressions so that (3+7) will be evaluated only once

```
[sqr sqr (3+7),(sqr (3+7)*sqr (3+7)),(sqr (3+7)*((3+7)*(3+7))), (sqr (3+7)*((3+7)*10
[sqr sqr (3+7),(sqr (3+7)*sqr (3+7)),(sqr(3+7)*((3+7)*(3+7))), (sqr (3+7)*((3+7)*10)
```

Lambda Calculus

Optional Topic

- M269 Unit 6/7 Reader *Logic and the Limits of Computation* alludes to other formalisations with equal power to a Turing Machine (pages 81 and 87)
- The *Reader* mentions Alonzo Church and his 1930s formalism (page 87, but does not give any detail)
- The notes in this section are optional and for comparison with the Turing Machine material
- Turing machine: explicit memory, state and implicit loop and case/if statement
- Lambda Calculus: function definition and application, explicit rules for evaluation (and transformation) of expressions, explicit rules for substitution (for function application)
- Lambda calculus reduction workbench
- Lambda Calculus Calculator

Lambda Calculus

Lambda Terms

- A **variable**, x, is a lambda term
- If M is a lambda term and x is a variable, then $(\lambda x.M)$ is a lambda term — a **lambda abstraction** or function definition
- If M and N are lambda terms, the $(M\ N)$ is lambda term — an **application**
- Nothing else is a lambda term

Lambda Calculus

Lambda Terms — Notational Conveniences

- Outermost parentheses are omitted $(M\ N) \equiv M\ N$
- Application is left associative $((M\ N)\ P) \equiv M\ N\ P$
- The body of an abstraction extends as far right as possible, subject to scope limited by parentheses
- $\lambda x.M\ N \equiv \lambda x.(M\ N)$ and not $(\lambda x.M)\ N$
- $\lambda x.\lambda y.\lambda z.M \equiv \lambda x\, y\, z.M$

Lambda Calculus

Lambda Calculus Semantics

- What do we mean by *evaluating an expression* ?
- To evaluate $(\lambda x.M)N$
- Evaluate M with x replaced by N
- This rule is called β-reduction
- $(\lambda x.M)N \underset{\beta}{\rightarrow} M[x := N]$
- $M[x := N]$ is M with occurrences of x replaced by N
- This operation is called *substitution* — see rules below

Lambda Calculus

β-Reduction Examples

- $(\lambda x.x)z \rightarrow z$
- $(\lambda x.y)z \rightarrow y$
- $(\lambda x.x\ y)z \rightarrow z\ y$

  a function that applies its argument to y
- $(\lambda x.x\ y)(\lambda z.z) \rightarrow (\lambda z.z)y \rightarrow y$
- $(\lambda x.\lambda y.x\ y)z \rightarrow \lambda y.z\ y$

  A *curried* function of two arguments — applies first argument to second
- *currying* replaces $f(x, y)$ with $(f\ x)y$ — nice notational convenience — gives *partial application* for free

Lambda Calculus

Substitution

- To define *substitution* use recursion on the structure of terms
- $x[x := N] \equiv N$
- $y[x := N] \equiv y$
- $(P\,Q)[x := N] \equiv (P[x := N])\,(Q[x := N])$
- $(\lambda x.M)[x := N] = \lambda x.M$

  In $(\lambda x.M)$, the x is a formal parameter and thus a local variable, different to any other
- $(\lambda y.M)[x := N] =$ what?
- Look back at the school maths example above — a subtle point

Lambda Calculus

Substitution (2)

- Renaming *bound variables consistently is allowed*
- $\lambda x.x \equiv \lambda y.y \equiv \lambda z.z$
- $\lambda y.\lambda x.y \equiv \lambda z.\lambda x.z$
- This is called *α-conversion*
- $(\lambda x.\lambda y.x\,y)\,y \to (\lambda x.\lambda z.x\,z)\,y \to \lambda z.y\,z$

Lambda Calculus

Substitution (3)

- **Bound and Free Variables**
- $BV(x) = \emptyset$
- $BV(\lambda x.M) = BV(M) \cup \{x\}$
- $BV(M\,N) = BV(M) \cup BV(N)$
- $FV(x) = \{x\}$
- $FV(\lambda x.M) = FV(M) - \{x\}$
- $FV(M\,N) = FV(M) \cup FV(N)$
- The above is a formalisation of school maths
- A Lambda term with no free variables is said to be *closed* — such terms are also called **combinators** — see Combinator and Combinatory logic (Hankin, 2004, page 10)
- **α-conversion**
- $\lambda x.M \underset{\alpha}{\rightarrow} \lambda y.M[x := y]$ if $y \notin FV(M)$

Lambda Calculus

Substitution (4)

- β-reduction final rule
- $(\lambda y.M)[x := N] = \lambda y.M$ if $x \notin FV(M)$
- $(\lambda y.M)[x := N] = \lambda y.M[x := N]$

  if $x \in FV(M)$ and $y \notin FV(N)$
- $(\lambda y.M)[x := N] = \lambda z.M[y := z][x := N]$

  if $x \in FV(M)$ and $y \in FV(N)$

  z is chosen to be first variable $z \notin FV(N\,M)$
- This is why you cannot go $f(a)$ when given
- $f(x) = ax^2 + bx + c$
- School maths — but made formal

Lambda Calculus

Rules Summary — Conversion

- α-**conversion** renaming bound variables
- $\lambda x.M \underset{\alpha}{\rightarrow} \lambda y.M[x := y]$ if $y \notin FV(M)$
- β-**conversion** function application
- $(\lambda x.M)N \underset{\beta}{\rightarrow} M[x := N]$
- η-**conversion** extensionality
- $\lambda x.F\,x \underset{\eta}{\rightarrow} F$ if $x \notin FV(F)$

Lambda Calculus

Rules Summary — Substitution

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$
3. $(P\,Q)[x := N] \equiv (P[x := N])\,(Q[x := N])$
4. $(\lambda x.M)[x := N] = \lambda x.M$
5. $(\lambda y.M)[x := N] = \lambda y.M$ if $x \notin FV(M)$
6. $(\lambda y.M)[x := N] = \lambda y.M[x := N]$
   if $x \in FV(M)$ and $y \notin FV(N)$
7. $(\lambda y.M)[x := N] = \lambda z.M[y := z][x := N]$
   if $x \in FV(M)$ and $y \in FV(N)$
   z is chosen to be first variable $z \notin FV(N\,M)$

Lambda Calculus

Lambda Calculus Encodings

- So what does this formalism get us ?
- The Lambda Calculus is Turing complete
- We can encode any computation (if we are clever enough)
- Booleans and propositional logic
- Pairs
- Natural numbers and arithmetic
- Looping and recursion

Lambda Calculus Encodings

Booleans and Propositional Logic

- True $= \lambda x.\lambda y.x$
- False $= \lambda x.\lambda y.y$
- IF a THEN b ELSE $c \equiv a\,b\,c$
- IF True THEN b ELSE $c \rightarrow (\lambda x.\lambda y.x)\,b\,c$
- $\rightarrow (\lambda y.b)\,c \rightarrow b$
- IF False THEN b ELSE $c \rightarrow (\lambda x.\lambda y.y)\,b\,c$
- $\rightarrow (\lambda y.y)\,c \rightarrow c$

Lambda Calculus Encodings

Booleans and Propositional Logic (2)

- Not $= \lambda x.((x\,\text{False})\text{True})$
- Not x = IF x THEN False ELSE True
- Exercise: evaluate Not True
- And $= \lambda x.\lambda y.((x\,y)\ \text{False})$
- And x y = IF x THEN y ELSE False
- Exercise: evaluate And True False
- Or $= \lambda x.\lambda y.((x\ \ \text{True}\ )\ y)$
- Or x y = IF x THEN True ELSE y
- Exercise: evaluate Or False True

Lambda Calculus Encodings

Booleans and Propositional Logic (2) — Exercises

- Exercise: evaluate Not True
- $\rightarrow (\lambda x.((x \ \text{False}) \ \text{True})) \ \text{True}$
- $\rightarrow (\text{True} \ \ \text{False}) \ \text{True}$
- Could go straight to False from here, but we shall fill in the detail
- $\rightarrow ((\lambda x.\lambda y.x)\,(\lambda x.\lambda y.y))\,(\lambda x.\lambda y.x)$
- $\rightarrow (\lambda y.(\lambda x.\lambda y.y))\,(\lambda x.\lambda y.x)$
- $\rightarrow (\lambda x.\lambda y.y) \equiv$ False
- Exercise: evaluate And True False
- $\rightarrow$(IF x THEN y ELSE False) True False
- $\rightarrow$(IF True THEN False ELSE False) $\rightarrow$False
- Exercise: evaluate Or False True
- $\rightarrow$(IF x THEN True ELSE y) False True
- $\rightarrow$(IF False THEN True ELSE True) $\rightarrow$True

Lambda Calculus Encodings

Natural Numbers — Church Numerals

- Encoding of natural numbers
- $0 = \lambda f.\lambda y.y$
- $1 = \lambda f.\lambda y.f\ y$
- $2 = \lambda f.\lambda y.f\ (f\ y)$
- $3 = \lambda f.\lambda y.f\ (f\ (f\ y))$
- Successor Succ $= \lambda z.\lambda f.\lambda y.f(z\ f\ y)$
- Succ $0 = (\lambda z.\lambda f.\lambda y.f(z\ f\ y))(\lambda f.\lambda y.y)$
- $\to \lambda f.\lambda y.f\ ((\lambda f.\lambda y.y)\ f\ y)$
- $\to \lambda f.\lambda y.f\ ((\lambda y.y)\ y)$
- $\to \lambda f.\lambda y.f\ y = 1$

Lambda Calculus Encodings

Natural Numbers — Operations

- isZero $= \lambda z.z(\lambda y.\text{ False })\text{ True}$
- Exercise: evaluate isZero 0
- If M and N are numerals (as λ expressions)
- Add $M\,N = \lambda x.\lambda y.(M\,x)\,((N\,x)\,y)$
- Mult $M\,N = \lambda x.(M\,(N\,x))$
- Exercise: show $1 + 1 = 2$

Lambda Calculus Encodings
Pairs

- Encoding of a pair a, b
- $(a, b) = \lambda x.$ IF x THEN a ELSE b
- FST $= \lambda f.f$ True
- SND $= \lambda f.f$ False
- Exercise: evaluate FST (a, b)
- Exercise: evaluate SND (a, b)

Lambda Calculus Encodings

The Fixpoint Combinator

- $Y = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$
- $Y\,F = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F$
- $\rightarrow (\lambda x.F\,(x\,x))(\lambda x.F\,(x\,x))$
- $F((\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))) = F\,(Y\,F)$
- $(Y\,F)$ is a fixed point of F
- We can use Y to achieve recursion for F

Lambda Calculus Encodings

The Fixpoint Combinator — Recursion

- **Recursion implementation — Factorial**
- $\text{Fact} = \lambda f.\lambda n.\ \text{IF}\ n = 0\ \text{THEN}\ 1\ \text{ELSE}\ n * (f\,(n - 1))$
- $(Y\ \text{Fact})1 = (\text{Fact}\ (Y\ \text{Fact}))1$
- $\rightarrow\ \text{IF}\ 1 = 0\ \text{THEN}\ 1\ \text{ELSE}\ 1 * ((Y\ \text{Fact})\,0)$
- $\rightarrow 1 * ((Y\ \text{Fact})\,0)$
- $\rightarrow 1 * (\text{Fact}\ (Y\ \text{Fact})\,0)$
- $\rightarrow 1 *\ \text{IF}\ 0 = 0\ \text{THEN}\ 1\ \text{ELSE}\ 0 * ((Y\ \text{Fact})\,(0 - 1))$
- $\rightarrow 1 * 1 \rightarrow 1$
- Factorial n = (Y Fact) n
- Recursion implemented with a non-recursive function Y

Computability

Turing Machines, Lambda Calculus and Programming Languages

- Anything computable can be represented as TM or Lambda Calculus
- But programs would be slow, large and hard to read
- In practice use the ideas to create more expressive languages which include built-in primitives
- Also leads to ideas on data types
- Polymorphic data types
- Algebraic data types
- Also leads on to ideas on higher order functions — functions that take functions as arguments or returns functions as results.

2 Complexity

- Complexity Classes **P** and **NP**
- Class **NP**
- NP-completeness
- NP-completeness and Boolean Satisfiability

Complexity

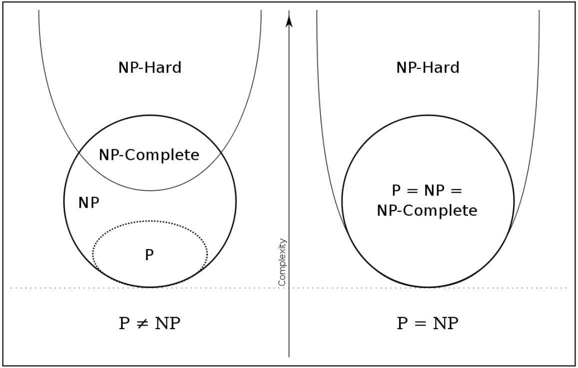
P and NP

- P, the set of all decision problems that can be solved in polynomial time on a deterministic Turing machine
- NP, the set of all decision problems whose solutions can be verified (certificate) in polynomial time
- Equivalently, NP, the set of all decision problems that can be solved in polynomial time on a non-deterministic Turing machine
- A *decision problem*, *dp* is *NP-complete* if
  1. *dp* is in NP and
  2. Every problem in NP is reducible to *dp* in polynomial time
- NP-hard — a problem satisfying the second condition, whether or not it satisfies the first condition. Class of problems which are at least as hard as the hardest problems in NP. NP-hard problems do not have to be in NP and may not be decision problems

Complexity

P and NP — Diagram

[Euler diagram](#) for P, NP, NP-complete and NP-hard set of problems



Source: Wikipedia NP-complete entry

Class NP

Certificate and Verifier

- To formalise the definition of the class **NP**, we need to formalise the idea of checking a candidate solution
- Define a *certificate* for each problem input that would return Yes
- Describe the *verifier* algorithm
- Demonstrate the *verifier* algorithm has polynomial complexity
- The terms *certificate* and *verifier* have technical definitions in terms of languages and Turing Machines but can be thought of as *candidate solution* and *checker algorithm*

Class **NP**

Example Decision Problems (1)

- **Composite Numbers** Given a number N decide if N is a composite (i.e. non-prime) number
  Certificate factorization of N
- **Connectivity** Given a graph G and two vertices s, t in G, decide if s is connected to t in G.
  Certificate path from s to t
- **Linear Programming** Given a list of m linear inequalities with rational coefficients over n variables $u_1, \ldots, u_n$ (a linear inequality has the form $a_1 u_1 + a_2 u_2 \cdots + a_n u_n \leqslant b$ for some coefficients $a_1, \ldots, a_n b$), decide if there is an assignment of rational numbers to the variables $u_1, \ldots, u_n$ which satisfies all the inequalities
  Certificate is the assignment

Class **NP**

Example Decision Problems (2)

- The above are in **P**
- *Composite Numbers*, *Connectivity* and *Linear programming* are in **P**
- *Composite Numbers* follows from Integer factorization and the AKS primality test from 2004
- *Connectivity* follows from the breadth-first search algorithm
- *Linear programming* shown to be in **P** by the Ellipsoid method

Class NP

Example Decision Problems (3)

- **Integer Programming** some or all variables are restricted to be integers
- **Travelling Salesperson** Given a set of nodes and distances between all pairs of nodes and a number k, decide if there is a closed circuit that visits every node exactly once and has total length at most k
  Certificate sequence of nodesin such a tour
- **Subset sum** Given a list of numbers and a number T, decide if there is a subset that adds up to T
  Certificate list of members of such a subset
- **Independent set (graph theory)** A subgraph of G with of at least k vertices which have no edges between them
  Certificate the list of k vertices
- **Clique problem** Given a graph and a number k, decide if there is a complete subgraph (clique) of size k
  Certificate list pf nodes. For explanation see Prove Clique is NP

Class **NP**

Example Decision Problems (4)

- The above are **NP-complete** — see List of NP-complete problems
- The following two are not known to be **P** nor **NP-complete**
- **Graph Isomorphism** Given two $n \times n$ adjacency matrices M_1, M_2, decide if M_1 and M_2 define the same graph (up to renaming of the vertices)
  Certificate the permutation $\pi : [n] \rightarrow [n]$ such that M_2 is equal to M_1 after reordering the indices of M_1 according to π
- **Integer factorization** Given three numbers N, L, U decide if N has a prime factor p in the interval $[L, U]$
  Certificate is the factorization of N

  Source Arora and Barak (2009, page 49) *Computational Complexity: A Modern Approach* and contained links

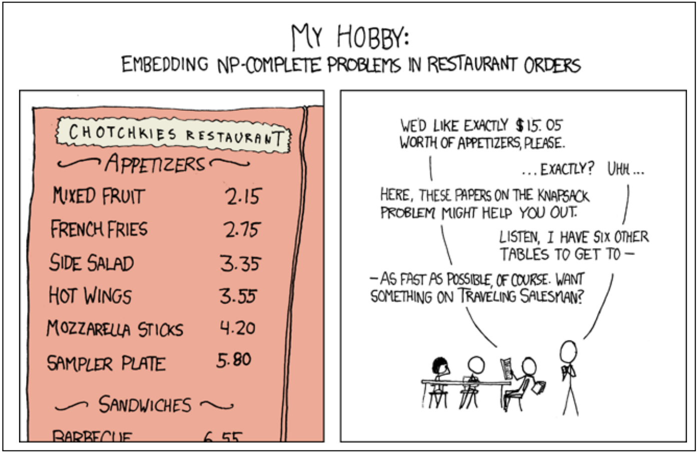
Complexity

NP-complete problems

- Boolean satisfiability (SAT) Cook-Levin theorem
- Conjunctive Normal Form 3SAT
- Hamiltonian path problem
- Travelling salesman problem
- NP-complete — see list of problems

Complexity

Knapsack Problem



Source & Explanation: XKCD 287

NP-Completeness and Boolean Satisfiability

Points on Notes

- The *Boolean satisfiability problem (SAT)* was the first decision problem shown to be *NP-Complete*
- This section gives a sketch of an explanation
- **Health Warning** different texts have different notations and there will be some inconsistency in these notes
- **Health warning** these notes use some formal notation *to make the ideas more precise* — computation requires precise notation and is about manipulating strings according to precise rules.

NP-Completeness and Boolean Satisfiability

Alphabets, Strings and Languages

- Notation:
- Σ is a set of symbols — the alphabet
- Σ^k is the set of all string of length k, which each symbol from Σ
- Example: if $\Sigma = \{0, 1\}$
  - $\Sigma^1 = \{0, 1\}$
  - $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^0 = \{\epsilon\}$ where ϵ is the empty string
- Σ^* is the set of all possible strings over Σ
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- A *Language*, L, over Σ is a subset of Σ^*
- $L \subseteq \Sigma^*$

NP-Completeness and Boolean Satisfiability

Language Accepted by a Turing Machine

- Language accepted by Turing Machine, M denoted by $L(M)$
- $L(M)$ is the set of strings $w \in \Sigma^*$ accepted by M
- For *Final States* $F = \{Y, N\}$, a string $w \in \Sigma^*$ is accepted by $M \Leftrightarrow$ (if and only if) M starting in q_0 with w on the tape halts in state Y
- Calculating a function (function problem) can be turned into a decision problem by asking whether $f(x) = y$

NP-Completeness and Boolean Satisfiability

The NP-Complete Class

- If we do not know if P $\neq$ NP, what can we say ?
- A language L is *NP-Complete* if:
  - $L \in$ NP and
  - for all other $L' \in$ NP there is a *polynomial time transformation* (Karp reducible, reduction) from L' to L
- Problem P_1 *polynomially reduces* (Karp reduces, transforms) to P_2, written $P_1 \propto P_2$ or $P_1 \leq_p P_2$, iff $\exists f : \mathrm{dp}_{P_1} \rightarrow \mathrm{dp}_{P_2}$ such that
  - $\forall I \in \mathrm{dp}_{P_1} [I \in Y_{P_1} \Leftrightarrow f(I) \in Y_{P_2}]$
  - f can be computed in polynomial time

NP-Completeness and Boolean Satisfiability

The NP-Complete Class (2)

- More formally, $L_1 \subseteq \Sigma_1^*$ polynomially transforms to $L_2 \subseteq \Sigma_2^*$, written $L_1 \propto L_2$ or $L_1 \leq_p L_2$, iff $\exists f : \Sigma_1^* \rightarrow \Sigma_2^*$ such that
  - $\forall x \in \Sigma_1^* [x \in L_1 \Leftrightarrow f(x) \in L_2]$
  - There is a polynomial time TM that computes f
- *Transitivity* If $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$
- If L is NP-Hard and $L \in$ P then P = NP
- If L is NP-Complete, then $L \in$ P if and only if P = NP
- If L_0 is NP-Complete and $L \in$ NP and $L_0 \propto L$ then L is NP-Complete
- Hence if we find one NP-Complete problem, it may become easier to find more
- In 1971/1973 Cook-Levin showed that the Boolean satisfiability problem (SAT) is NP-Complete

NP-Completeness and Boolean Satisfiability

The Boolean Satisfiability Problem

- A propositional logic formula or Boolean expression is built from variables, operators: AND (conjunction, $\wedge$), OR (disjunction, $\vee$), NOT (negation, $\neg$)
- A formula is said to be *satisfiable* if it can be made True by some assignment to its variables.
- *The Boolean Satisfiability Problem* is, given a formula, check if it is satisfiable.
  - *Instance:* a finite set U of Boolean variables and a finite set C of clauses over U
  - *Question:* Is there a satisfying truth assignment for C ?
- A *clause* is is a disjunction of variables or negations of variables
- *Conjunctive normal form (CNF)* is a conjunction of clauses
- Any Boolean expression can be transformed to CNF

NP-Completeness and Boolean Satisfiability

The Boolean Satisfiability Problem (2)

- Given a set of Boolean variable $U = \{u_1, u_2, \ldots, u_n\}$
- A literal from U is either any u_i or the negation of some u_i (written $\overline{u_i}$) usual notation $\neg u_i$
- A clause is denoted as a subset of literals from U — $\{u_2, \overline{u_4}, u_5\}$ usual notation $u_2 \lor \neg u_4 \lor u_5$
- A clause is satisfied by an assignment to the variables if at least one of the literals evaluates to True (just like disjunction of the literals)
- Let C be a set of clauses over U — C is satisfiable iff there is some assignment of truth values to the variables so that every clause is satisfied (just like CNF)
- $C = \{\{u_1, u_2, u_3\}, \{\overline{u_2}, \overline{u_3}\}, \{u_2, \overline{u_3}\}\}$ is satisfiable

  usual notation $(u_1 \lor u_2 \lor u_3) \land (\neg u_2 \lor \neg u_3) \land (u_2 \lor \neg u_3)$

  assign $(u_1, u_2, u_3) = (T, F, F), (T, T, F), (F, T, F)$
- $C = \{\{u_1, u_2\}, \{u_1, \overline{u_2}\}, \{\overline{u_1}\}\}$ is not satisfiable

  usual notation $(u_1 \lor u_2) \land (u_1 \lor \neg u_2) \land (\neg u_1)$

NP-Completeness and Boolean Satisfiability

The Boolean Satisfiability Problem (3)

- Proof that SAT is NP-Complete looks at the structure of NDTMs and shows you can transform any NDTM to SAT in polynomial time (in fact logarithmic space suffices)
- SAT is in NP since you can check a solution in polynomial time
- To show that $\forall L \in \text{NP} : L \propto \text{SAT}$ invent a polynomial time algorithm for each polynomial time NDTM, M, which takes as input a string x and produces a Boolean formula E_x which is satisfiable iff M accepts x
- See Cook-Levin theorem

NP-Completeness and Boolean Satisfiability

Coping with NP-Completeness

- What does it mean if a problem is NP-Complete ?
  - There is a P time verification algorithm.
  - There is a P time algorithm to solve it iff P = NP (?)
  - No one has yet found a P time algorithm to solve any NP-Complete problem
  - So what do we do ?
- Improved exhaustive search — Dynamic Programming; Branch and Bound
- Heuristic methods — *acceptable* solutions in *acceptable* time — compromise on optimality
- Average time analysis — look for an algorithm with good average time — compromise on generality (see Big-O Algorithm Complexity Cheatsheet)
- Probabilistic or Randomized algorithms — compromise on correctness

What Next ?

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

What Next ?

To err is human ?

- 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)

Future Work

Dates

- Sunday, 4 May 2026 online tutorial TMA03 topics
- Thursday, 22 May 2025 TMA03 due

Web Sites

Computability

- **Logic**
  - WFF, WFF'N Proof online
- **Computability**
  - Computability
  - Computable function
  - Decidability (logic)
  - Turing Machines
  - Universal Turing Machine
  - Turing machine simulator
  - Lambda Calculus
  - Von Neumann Architecture
  - Turing Machine XKCD 205 Candy Button Paper
  - Turing Machine XKCD 505 A Bunch of Rocks
  - RIP John Conway Why can Conway's Game of Life be classified as a universal machine?
  - Phil Wadler Bright Club on Computability
  - Bridges: Theory of Computation: Halting Problem
  - Bridges: Theory of Computation: Other Non-computable Problems

Web Sites

Complexity

- **Complexity**
  - Complexity class
  - NP complexity
  - NP complete
  - Reduction (complexity)
  - P versus NP problem
  - Graph of NP-Complete Problems